



Java微服务实战

本书由浅入深地讲解了微服务的相关技术，包括基础框架、服务框架和监控部署三大部分，以实战为主、理论为辅，内容丰富，实用性强。

赵计刚 / 著



中国工信出版集团



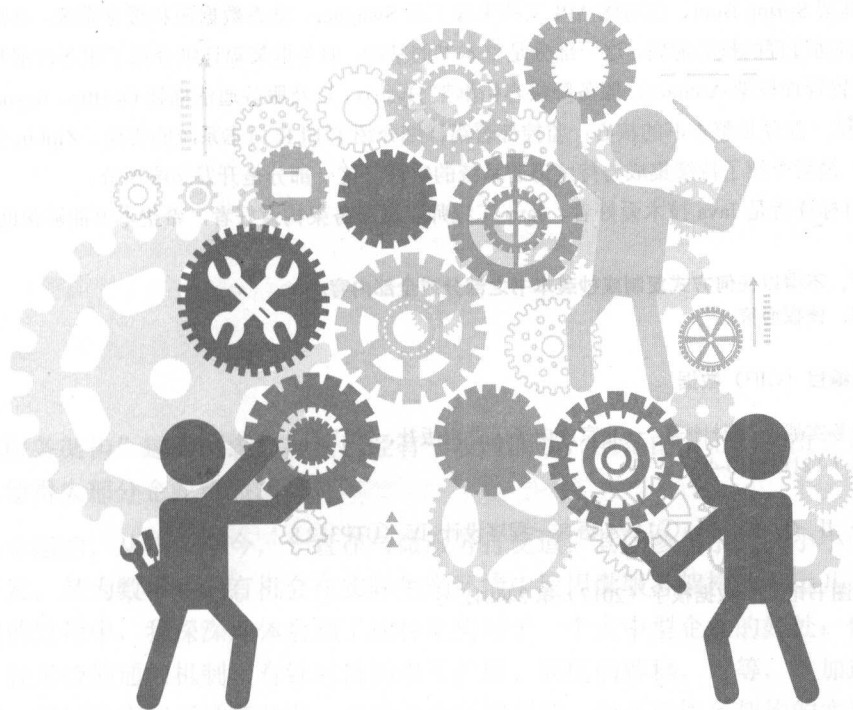
电子工业出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY
http://www.phei.com.cn

关于作者



赵计刚

现任网易高级Java开发工程师。毕业于哈尔滨工业大学软件学院。2016年3月加入51信用卡，开始接触微服务架构，之后一直从事微服务的开发与研究，学习与总结了不少微服务架构相关的理论与实践经验。个人是开源技术的拥趸，对新技术充满浓厚的兴趣，尤其是微服务架构相关技术。



Java微服务实战

赵计刚 / 著

电子工业出版社

Publishing House of Electronics Industry

北京·BEIJING

内 容 简 介

本书分为三部分：基础框架篇（第 1~6 章）、服务框架篇（第 7~10 章）、监控部署篇（第 11~13 章），由浅入深地讲解了微服务的相关技术。基础框架篇从微服务架构的基本概念与技术选型出发，详细介绍了微服务基础框架 Spring Boot、自动化 API 文档生成工具 Swagger、动态数据源和缓存系统，并深入分析了 Spring Boot 启动过程的核心源码，这一部分是整本书的基础；服务框架篇详细介绍了服务注册与发现框架 Consul、热配置管理框架 Archaius、服务降级容错框架 Hystrix，以及服务通信框架 OkHttp、AsyncHttpClient 和 Retrofit，这一部分是整本书的核心；监控部署篇详细介绍了 ELK 日志系统的实现、Zipkin 全链路追踪系统的实现，最后介绍了持续集成与持续部署系统的实现，这一部分是开发运维部分。

本书的目标读者是 Java 技术爱好者、Java 工程师、微服务架构爱好者，希望本书能够帮助到你们。

未经许可，不得以任何方式复制或抄袭本书之部分或全部内容。
版权所有，侵权必究。

图书在版编目（CIP）数据

Java 微服务实战 / 赵计刚著. —北京：电子工业出版社，2017.11
ISBN 978-7-121-32840-4

I. ①J… II. ①赵… III. ①JAVA 语言—程序设计 IV. ①TP312.8

中国版本图书馆 CIP 数据核字（2017）第 244012 号

策划编辑：付 睿

责任编辑：牛 勇

印 刷：三河市良远印务有限公司

装 订：三河市良远印务有限公司

出版发行：电子工业出版社

北京市海淀区万寿路 173 信箱

邮编：100036

开 本：787×980 1/16

印张：18.75

字数：386 千字

版 次：2017 年 11 月第 1 版

印 次：2017 年 11 月第 1 次印刷

定 价：69.00 元

凡所购买电子工业出版社图书有缺损问题，请向购买书店调换。若书店售缺，请与本社发行部联系，联系及邮购电话：(010) 88254888, 88258888。

质量投诉请发邮件至 zltz@phei.com.cn，盗版侵权举报请发邮件至 dbqq@phei.com.cn。

本书咨询联系方式：010-51260888-819 faq@phei.com.cn。

前言

“微服务架构”这个概念的提出已经有一段时间了，但是由于资料的匮乏以及实现的复杂性，使得大部分企业望而却步。

我是幸运的，从毕业至今，一直在与微服务打交道，其间参与了大小多个微服务项目的开发，是为数不多的有机会在实际生产环境中运用微服务架构的幸运儿。在使用微服务架构的过程中，我深深地体会到了这种架构对于一个大中型企业的好处：快速的开发与部署、轻量级的通信机制、有针对性的水平扩展、高度的解耦，等等，这加速了一个项目的迭代，很好地实现了敏捷开发，正是企业所需要的。但是微服务架构的实现也是有一定的复杂性的：服务拆分的边界怎么来定义；原本的单机事务在服务拆分之后变成了分布式事务，这怎么处理；由于服务拆分了，服务之间的通信需要走网络，怎样尽可能地减少网络通信的消耗；怎样防止服务雪崩；怎么梳理链路调用关系，怎么快速定位导致调用链发生错误的服务；怎样监控服务的健康状态，等等，这都是使用了微服务架构后需要解决的问题。本书结合我在实际使用微服务架构中积累的经验给出了其中大多数问题的解决方案，可以为读者朋友提供一个参考。

不可否认的是，正如文章开头所讲的，国内关于微服务架构的开发学习资料与课程都非常有限，这使得微服务架构在国内的推广并没有想象中那样火热。而且，国内的相关资料大多数以理论为主或者没有实战基础。所以，当电子工业出版社博文视点的付睿老师提议写一本以实战为主的微服务书籍之后，我毫不犹豫地抓住了这个机会。本书以实战为主，以理论为辅，真正给出了能在实际生产中使用的技术方案。由于篇幅限制以及以实战为主的特点，本书不会介绍太多的理论（哪怕这个理论很重要），比如在介绍 Consul 的时候，本书不会详细介绍 Raft 一致性协议，但是会介绍与其相关的一些在使用中需要注意的问题，

如果读者对相关问题有兴趣，可以查看相关的论文资料。

本书的组织结构

本书从组织结构上来讲，分为三部分：基础框架篇（第 1~6 章）、服务框架篇（第 7~10 章）、监控部署篇（第 11~13 章）。

第 1 章 微服务概述

本章首先介绍了微服务架构的概念与优缺点，之后简略介绍了微服务中需要的各种组件与常见的技术选型。

第 2 章 微服务基础框架

本章首先介绍了 Spring Boot 在微服务方面的优势，之后通过从零开始开发一个 Spring Boot 项目来介绍 Spring Boot 的基本使用方法，使没有使用过 Spring Boot 的同学可以快速入门。最后在“再学一招”部分，介绍了一个非常好用的 Maven 命令：Maven 依赖树，该命令是查看 SpringBoot-Starter 的依赖以及处理依赖冲突的利器。

第 3 章 微服务文档输出

本章首先介绍了自动化文档输出工具 Swagger 的概念，之后介绍了 Swagger 与 Spring Boot 的集成以及 Swagger 的常用注解。最后在“再学一招”部分，介绍了一个很好用的消除模板代码的框架 Lombok 的安装与使用方法。

第 4 章 微服务数据库

本章以 MySQL 为例，首先介绍了在单数据源的情况下，Spring Boot 与 MyBatis 的集成。之后使用 AbstractRoutingDataSource 实现了对多数据源情况的处理，并简要介绍了实现多数据源的原理。最后在“再学一招”部分，介绍了 MyBatis-Generator 的基本用法。

第 5 章 微服务缓存系统

本章首先介绍了常用的缓存技术的优缺点与选型方案，之后介绍了当使用 Redis 2.x 版本时，使用 Spring Boot 集成 ShardedJedis 实现客户端分片的方法。然后介绍了 Redis 3.x 集群的搭建与使用 Spring Boot 集成 JedisCluster 实现服务端集群的方法。最后简要分析了 JedisCluster 的源码。在本章的“再学一招”部分，介绍了使用 GuavaCache 实现本地缓存的方法。

第 6 章 Spring Boot 启动源码解析

本章详细分析了 Spring Boot 启动过程的源码，掌握这一章对于后续章节的学习至关重要。在本章的“再学一招”部分，简要介绍了在开发过程中获取配置信息的 4 种方法。

第 7 章 微服务注册与发现

本章首先介绍了 Consul 的基本概念和功能，之后搭建了服务提供者和服务调用者两个项目来实现使用 Consul 进行服务注册和服务发现的功能，最后介绍了使用 Consul 与 SpringBoot-Actuator 实现服务健康检查的功能。在本章的“再学一招”部分，简要介绍了 Consul 自身提供的几种健康检查的方式及原理。

第 8 章 微服务配置管理

本章首先介绍了为什么要使用 Archaius 以及 Archaius 实现服务热配置的原理，之后展示了使用 Consul-KV 实现配置中心的方式以及结合 Archaius 实现配置动态获取的方式，最后提供了一种将 Archaius 配置信息与 Spring 的 PropertySource 结合的方案。在本章的“再学一招”部分，笔者详细分析了使用 Archaius 构造动态属性源以及动态获取属性的源码。

第 9 章 微服务进程间通信

本章首先介绍了三种服务通信框架：OkHttp、AsyncHttpClient 和 Retrofit，之后分别展示了使用三种框架进行服务通信的代码编写方法。最后在本章的“再学一招”部分，详细分析了使用 Retrofit 进行服务通信的核心源码。

第 10 章 微服务降级容错

本章首先详细介绍了为什么使用 Hystrix、Hystrix 的工作原理以及执行流程，之后展示了在实际项目中如何使用 Hystrix 实现服务降级容错，最后展示了怎样结合 Turbine 来搭建一个完整的 Hystrix 监控系统。在本章的“再学一招”部分，介绍了设置 Hystrix 配置参数的两种方法以及最常使用的 11 个配置项。

第 11 章 微服务日志系统

本章首先详细介绍了为什么使用 ELK 以及 ELK 最常用的两种架构，之后搭建了 ELK 缓冲系统，然后展示了怎样将项目中的日志发送到日志系统中，最后简单介绍了 Kibana 的常见用法。在本章的“再学一招”部分，介绍了怎样使用 Elasticsearch-Curator 进行日志的定时删除。

第 12 章 微服务全链路追踪系统

本章首先详细介绍了为什么使用 Zipkin、Zipkin 的工作流程、数据模型以及工作原理，之后搭建了 Zipkin 全链路追踪系统，然后分别展示了使用 AsyncHttpClient 和 OkHttp 实现服务通信时进行链路追踪的方式，并且介绍了将追踪信息进行持久化的方法。在本章的“再学一招”部分，详细分析了 Brave（Zipkin 的官方 Java 客户端）的核心源码。

第 13 章 微服务持续集成与持续部署系统

本章首先详细介绍了为什么需要搭建持续集成与持续部署系统，之后介绍了构建这套系统的技术选型：GitLab、Jenkins、Docker-Registry 与总体架构，然后分别介绍了使用 jar 包部署服务和使用 Docker 镜像部署服务时持续集成与持续部署系统的工作原理。之后，搭建了这套系统，最后分别展示了在使用 jar 包部署服务和使用 Docker 镜像部署服务时，持续集成与持续部署系统的实现方式。在本章的“再学一招”部分，介绍了最常用的 10 条 Docker 命令。

目标读者

本书面向的读者群：

- Java 技术爱好者
- Java 工程师
- 微服务架构爱好者

本书特点

- 以实战为主，理论为辅，代码编写占了绝大部分的篇幅。
- 代码由浅到深，会介绍表层代码下的核心源码实现。
- 除了第 1 章外，每章的结尾都会提供一个“再学一招”部分，介绍好用的技术或者解析源码。

勘误与支持

由于作者经验水平有限，书中难免有错漏之处。在本书出版后的任何时间，若您对本书有任何问题，可以发邮件到 1197596604@qq.com，我会对所有问题给予回复。也可以加

入 QQ 群 341027254 进行技术交流!

致谢

感谢 51 信用卡, 感谢你为我提供这样一个平台, 让我能够学习到很多感兴趣的技术, 并能将这些技术应用到实际的项目中。也感谢在 51 信用卡中并肩作战的伙伴们, 能在这里与一群牛人一起工作让我感到非常自豪。

感谢 51 信用卡首席架构师孔晨, 你敢为人先的精神和雄厚的技术沉淀使得在国内并不盛行的微服务架构在 51 信用卡内部运行得炉火纯青。也感谢你不辞辛劳的指导, 打通了我在技术道路上的诸多症结。

感谢电子工业出版社博文视点的付睿老师, 如果没有你的提议和引导, 就不会有这本书, 你严谨认真的工作态度让我非常敬佩。

感谢我亲爱的老婆, 是你的支持与谅解, 才会让我能够有足够多的时间来完成这本书!

谨以此书献给我最敬佩的技术人员、最亲爱的家人, 以及众多热爱微服务的朋友们!

目录

第 1 篇 基础框架篇

第 1 章 微服务概述	2
1.1 初识微服务	2
1.1.1 什么是微服务	2
1.1.2 为什么需要微服务	3
1.1.3 微服务架构的缺点	4
1.2 微服务中的组件与技术选型	5
第 2 章 微服务基础框架	11
2.1 Spring Boot 的优势	11
2.2 Spring Boot 入门	11
2.2.1 搭建项目框架	11
2.2.2 开发第一个 Spring Boot 程序	12
2.2.3 运行 Spring Boot 项目	15
2.3 再学一招：使用 Maven 依赖树验证 Spring Boot 自动引包功能	16
第 3 章 微服务文档输出	18
3.1 Swagger 概述	18
3.2 如何使用 Swagger	18

3.2.1	搭建项目框架	18
3.2.2	Spring Boot 集成 Swagger	19
3.2.3	分析 Swagger 生成的 API 文档	24
3.2.4	使用 Swagger 进行接口调用	24
3.3	再学一招：使用 Lombok 消除 POJO 类模板代码	25
第 4 章	微服务数据库	27
4.1	单数据源	27
4.1.1	搭建项目框架	27
4.1.2	建库和建表	28
4.1.3	使用 MyBatis-Generator 生成数据访问层	28
4.1.4	Spring Boot 集成 MyBatis	30
4.2	多数据源	39
4.2.1	建库和建表	40
4.2.2	使用 MyBatis-Generator 生成数据访问层	41
4.2.3	结合 AbstractRoutingDataSource 实现动态数据源	42
4.2.4	使用 AOP 简化数据源选择功能	48
4.2.5	实现多数据源的步骤总结	49
4.3	再学一招：MyBatis-Generator 基本用法	50
第 5 章	微服务缓存系统	53
5.1	常用的缓存技术	53
5.1.1	本地缓存与分布式缓存	53
5.1.2	Memcached 与 Redis	54
5.2	Redis 2.x 客户端分片	54
5.2.1	安装 Redis	54
5.2.2	Spring Boot 集成 ShardedJedis	55
5.3	Redis 3.x 集群	60
5.3.1	搭建 Redis 集群	60
5.3.2	Spring Boot 集成 JedisCluster	63
5.3.3	JedisCluster 关键源码解析	65
5.4	再学一招：使用 GuavaCache 实现本地缓存	67

第6章 Spring Boot 启动源码解析	70
6.1 创建 SpringApplication 实例	71
6.1.1 判断是否是 Web 环境	72
6.1.2 创建并初始化 ApplicationInitializer 列表	72
6.1.3 创建并初始化 ApplicationListener 列表	75
6.1.4 初始化主类 mainApplicationClass	76
6.2 添加自定义监听器	76
6.3 启动核心 run 方法	77
6.3.1 创建启动停止计时器	78
6.3.2 配置 awt 系统属性	79
6.3.3 获取 SpringApplicationRunListeners	80
6.3.4 启动 SpringApplicationRunListener	81
6.3.5 创建 ApplicationArguments	81
6.3.6 创建并初始化 ConfigurableEnvironment	82
6.3.7 打印 Banner	88
6.3.8 创建 ConfigurableApplicationContext	88
6.3.9 准备 ConfigurableApplicationContext	90
6.3.10 刷新 ConfigurableApplicationContext	92
6.3.11 容器刷新后动作	94
6.3.12 SpringApplicationRunListeners 发布 finish 事件	95
6.3.13 计时器停止计时	95
6.4 再学一招：常用的获取属性的 4 种方法	95

第2篇 服务框架篇

第7章 微服务注册与发现	98
7.1 初识 Consul	98
7.2 搭建 Consul 集群	99
7.2.1 安装 Consul	99
7.2.2 启动 Consul 集群	99
7.2.3 启动 Consul-UI	101
7.3 使用 Consul 实现服务注册与服务发现	102

7.3.1	搭建项目框架	102
7.3.2	配置服务注册信息	104
7.3.3	实现服务启动注册	106
7.3.4	实现服务发现	108
7.4	服务部署测试	110
7.4.1	编写测试类	110
7.4.2	服务打包部署	111
7.4.3	运行测试	113
7.5	使用 Consul 与 Actuator 实现健康检查	113
7.5.1	健康检查机制	113
7.5.2	健康检查查错思路	113
7.6	再学一招: Consul 健康检查分类及原理	114
第 8 章	微服务配置管理	116
8.1	初识 Archaius	116
8.1.1	为什么要使用 Archaius	116
8.1.2	Archaius 原理	116
8.2	使用 Consul-KV 实现配置集中管理	117
8.3	使用 Archaius 实现动态获取配置	118
8.3.1	搭建项目框架	118
8.3.2	创建配置信息读取源	120
8.3.3	实现服务启动时读取配置信息	122
8.3.4	动态获取配置信息	124
8.3.5	将配置信息动态加入 Spring 属性源的思路	125
8.4	再学一招: Archaius 关键源码解析	125
8.4.1	构造动态属性源	125
8.4.2	动态获取属性	129
第 9 章	微服务进程间通信	131
9.1	常见的三种服务通信技术	131
9.2	创建一个简单的被调用服务	132
9.2.1	搭建项目框架	132

9.2.2	实现一个简单的被调用接口	134
9.3	使用 OkHttp 实现服务通信	136
9.3.1	搭建项目框架	136
9.3.2	创建 OkHttp 调用实体类	137
9.3.3	实现服务通信功能	138
9.3.4	Spring Boot 指定服务启动端口的三种方式	140
9.4	使用 AsyncHttpClient 实现服务通信	141
9.4.1	搭建项目框架	141
9.4.2	创建 AsyncHttpClient 调用实体类	141
9.4.3	实现服务通信功能	142
9.5	使用 Retrofit 实现服务通信	143
9.5.1	搭建项目框架	143
9.5.2	创建调用接口并实例化接口	143
9.5.3	实现服务通信功能	145
9.6	再学一招: Retrofit 源码解析	145
9.6.1	构造 RestAdapter	146
9.6.2	初始化 RestAdapter.Builder 属性	148
9.6.3	创建 RestAdapter 实例	151
9.6.4	构造请求方法的接口类	152
9.6.5	校验 service 接口的合法性	153
9.6.6	使用动态代理创建对象	154
9.6.7	进行请求调用	154
9.6.8	获取 RestMethodInfo 实例	156
9.6.9	进行方法调用	156
9.6.10	加载 RestMethodInfo 的剩余属性	158
9.6.11	构建请求参数 retrofit.client.Request	162
9.6.12	利用 clientProvider 进行真正的调用	163
9.6.13	处理响应	164
第 10 章	微服务降级容错	165
10.1	初识 Hystrix	165
10.1.1	为什么要使用 Hystrix	165

10.1.2	Hystrix 工作原理	166
10.1.3	Hystrix 执行流程	168
10.2	使用 Hystrix 实现服务降级容错	169
10.2.1	搭建项目框架	169
10.2.2	创建 AsyncHttpClient 调用实体类	172
10.2.3	服务通信框架集成服务降级容错功能	173
10.2.4	验证服务降级容错功能	175
10.3	搭建 Hystrix 监控系统	178
10.3.1	使用 Hystrix-Metrics-Event-Stream 发布监控信息	178
10.3.2	使用 Hystrix-Dashboard 展示监控信息	179
10.3.3	使用 Turbine 聚合监控信息	181
10.4	再学一招: Hystrix 常用配置	186
10.4.1	设置配置参数的两种方法	186
10.4.2	常见配置项的配置方式	186

第 3 篇 监控部署篇

第 11 章	微服务日志系统	190
11.1	初识 ELK	190
11.1.1	为什么要用 ELK	190
11.1.2	ELK 最常用的两种架构	191
11.2	搭建 ELK 系统	192
11.2.1	安装配置启动 Redis	193
11.2.2	安装配置启动 Elasticsearch	193
11.2.3	安装配置启动 Logstash-Shipper	195
11.2.4	安装配置启动 Logstash-Indexer	197
11.2.5	安装配置启动 Kibana	198
11.3	使用 LogbackAppender 发送日志	199
11.3.1	搭建项目框架	199
11.3.2	配置 logback.xml 文件	201
11.3.3	创建 LogbackAppender 发送日志	202
11.3.4	验证日志输出查询功能	204

11.4	Kibana 常见用法	206
11.4.1	日期选择	206
11.4.2	自动刷新	207
11.4.3	查询语法	207
11.5	再学一招：使用 Curator 定时删除日志	208
11.5.1	安装 Curator	208
11.5.2	配置 Curator	209
11.5.3	配置 crontab 定时任务	211
11.5.4	验证定时任务	211
第 12 章	微服务全链路追踪系统	213
12.1	初识 Zipkin	213
12.1.1	为什么要使用 Zipkin	213
12.1.2	Zipkin 工作流程	214
12.1.3	Zipkin 数据模型	216
12.1.4	Zipkin 工作原理	216
12.2	使用 Zipkin 搭建全链路追踪系统	218
12.3	使用 Brave + AsyncHttpClient 实现全链路追踪	220
12.3.1	搭建项目框架	220
12.3.2	使用服务端拦截器补充追踪信息	222
12.3.3	使用客户端拦截器创建、销毁追踪信息	226
12.3.4	使用 Zipkin-webUI 查询链路追踪信息	228
12.4	使用 MySQL 持久化追踪信息	230
12.4.1	创建三张追踪信息表	230
12.4.2	使用 Brave-MySQL 存储追踪信息	233
12.5	使用 Brave-OkHttp 实现全链路追踪	233
12.5.1	搭建项目框架	234
12.5.2	使用服务端与客户端拦截器收集追踪信息	236
12.6	再学一招：Brave 关键源码解析	239
12.6.1	span 的生命周期	239
12.6.2	使用 reporter 创建 span	240
12.6.3	使用 collector 收集 span	245

12.6.4	使用 collector 发送 span	247
第 13 章 微服务持续集成与持续部署系统		251
13.1	初识持续集成与持续部署系统	251
13.2	系统总体架构	252
13.2.1	初识 GitLab	252
13.2.2	初识 Jenkins	253
13.2.3	初识 Docker-Registry	253
13.3	持续集成与持续部署系统工作原理	253
13.3.1	使用 jar 包部署项目的整体流程	253
13.3.2	使用 Docker 镜像部署项目的整体流程	253
13.4	搭建持续集成与持续部署系统	254
13.4.1	安装启动 Docker	254
13.4.2	安装配置启动 GitLab	255
13.4.3	安装启动 Jenkins	257
13.4.4	配置 Jenkins 全局信息与安装插件	259
13.4.5	安装配置启动 Docker-Registry	260
13.5	使用 jar 包方式部署服务	261
13.5.1	搭建项目框架	261
13.5.2	使用 GitLab 创建组和项目	263
13.5.3	使用 GitLab 管理代码	266
13.5.4	使用 Jenkins 编译打包服务	267
13.5.5	使用 webhook 实现服务的持续集成	270
13.5.6	使用 Jenkins + Shell 实现服务的持续部署	272
13.6	使用 Docker 镜像方式部署服务	276
13.6.1	搭建项目框架	276
13.6.2	编写 Dockerfile 文件创建镜像	278
13.6.3	通过 Jenkins + Shell 使用镜像实现持续部署	279
13.6.4	分析 Jenkins 构建日志	280
13.7	再学一招: Docker 常用命令	281

用 HTTP 资源接口。单个服务也可以通过对全自动化的部署机制来独立部署；微服务中的单个服务可以以多种语言来编写。但是在实际开发中，由于各个公司的技术栈有限，通常会指定一些特定的技术语言，例如 Java。每个服务都可以使用不同数据库存储技术，例如 MySQL、Cassandra、或 MongoDB 等。但是为了统一，通常各个服务还是会使用同一种存储技术。

第 2 章 为什么需要微服务

微服务架构

为什么需要微服务？通常会把微服务架构和单体架构进行比较，二者的架构图对比如

第 1 篇 基础框架篇



第 1 章 微服务概述

第 2 章 微服务基础框架

第 3 章 微服务文档输出

第 4 章 微服务数据库

第 5 章 微服务缓存系统

第 6 章 Spring Boot 启动源码解析

第 1 章

微服务概述

1.1 初识微服务

1.1.1 什么是微服务

“微服务”是软件架构大师 Martin Fowler 提出来的，但至今为止其都没有一个确切的定义。只有在 Martin Fowler 的官方网站上给出了一段关于微服务架构的说明。该段说明原文如下：

In short, the microservice architectural style is an approach to developing a single application as a suite of small services, each running in its own process and communicating with lightweight mechanisms, often an HTTP resource API. These services are built around business capabilities and independently deployable by fully automated deployment machinery. There is a bare minimum of centralized management of these services, which may be written in different programming languages and use different data storage technologies.

简单来讲，微服务架构风格是将一个单体的应用程序开发拆解为一组“小”的服务，值得注意的是，这里所说的“小”是以业务边界来区分的，而不是根据代码的多少来区分的。每个服务都运行在一个单独的进程中，服务之间通过轻量级的机制进行通信，例如使

用 **HTTP 资源接口**；每个服务都可以通过全自动化的部署机制来独立部署；微服务中的各个服务可以以多种语言来编写，但是在实际开发中，由于各个公司的技术栈有限，通常会指定一门擅长的技术语言，例如 Java；每个服务都可以使用不同数据存储技术，例如 MySQL、Cassandra 及 MongoDB 等，但是为了统一，通常各个服务还是会选用同一种存储技术。

1.1.2 为什么需要微服务

为什么需要微服务？通常会将微服务架构和单体架构进行比较。二者的架构图对比如图 1-1 所示。

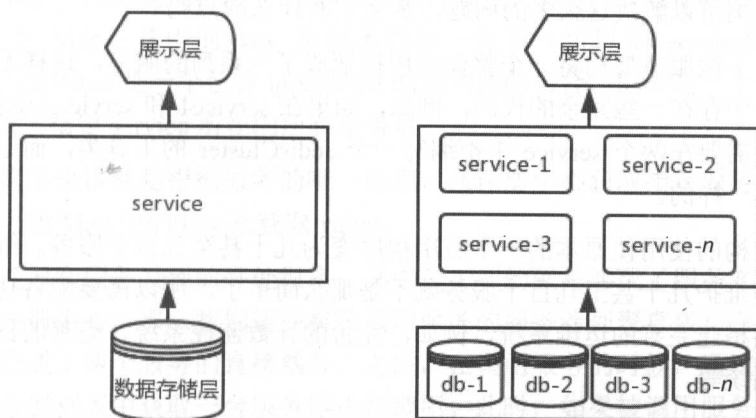


图1-1 单体架构与微服务架构图比较

左边是单体架构图，右边是微服务架构图。传统的单体架构主要包括三部分：一个展示层，用于将一些信息展示给客户端人员或者为客户端人员提供一些交互页面；一个数据存储层，通常就是提供一个数据库，用于存储一些需要持久化的数据；最后一个部分，是一个服务端的应用程序，该程序主要用于处理请求、执行业务逻辑、操作数据库，以及将相关结果返回给前端等。在该架构中，所有的请求都在一个进程中处理，而且水平扩展也很简单，只需要多添加几台部署了该服务的机器，之后在这些服务器的前边部署一台负载均衡器就可以了。

单体架构有这么几个问题：首先，由于所有的业务逻辑都写在了一个应用 `service` 中，因此只要对该 `service` 进行修改，哪怕只是添加一行代码，也需要编译打包部署整个应用，需要的时间会比较久，耗时耗力；其次，假设整个应用中只有一个接口到达了瓶颈，我们想要水平扩展该接口，这个时候只能通过水平扩展整个应用来达到目的；最后，随着应用

程序规模的增加，即使我们使用 Maven 进行模块化开发，也很难保证对于一个模块的修改不会影响到其他模块。

为了解决这些问题，微服务架构出现了！在微服务架构中，我们将一个整体应用切分成了多个小的服务，这些小服务可以独立部署并且每个服务运行在自己的进程中，所以如果修改了一个服务的代码，那么我们只需要单独部署该服务就行，而且如果需要水平扩展一个服务的接口，只扩展该服务就可以了。

1.1.3 微服务架构的缺点

微服务架构可以解决这么多的问题，那么它有什么缺点吗？

首先，由于微服务架构将一个整体应用拆解成了一系列的服务，这样不可避免地就会在不同的服务中存在一些冗余的代码，例如，如果在 service1 和 service2 中都用到了 Redis 集群，可能就需要在两个 service 中都编写一个 JedisCluster 的工具类，而这个工具类的代码几乎是一模一样的。

微服务架构的使用使原本的一个应用程序变为几十甚至几百个服务。维护一个应用程序简单，但是维护几十甚至几百个服务就不是那么简单了，所以需要配备功底较深的运维人员，也需要搭建完善的运维系统，例如，完善的计数监控系统、完善的日志系统及完善的链路跟踪系统等。

在微服务架构中，服务众多，服务之间的调用也由原来的单体架构中的进程内调用变为了进程之间调用，这就需要考虑很多问题，例如使用什么通信方式，如何处理网络延迟及服务容错等问题。这是分布式系统都会存在的问题，但是在微服务架构中这些问题尤其严重，因为“分布”得非常厉害。既然是分布式系统，就还会存在一个很大的问题：分布式事务的处理。分布式事务的解决方式虽然比较多，但是几乎没有十全十美的方式。通常会使用补偿的方式。

另外，如果从零开始搭建一套微服务系统，那么需要掌握的组件技术会比较多，从而开发的难度较大，开发周期会比较长。在开源世界中有一个比较好的微服务框架：Spring Cloud。但是如果使用 Spring Cloud，定制性就远不如自己搭建的微服务系统。另外，即使使用 Spring Cloud，一些系统还是需要自己搭建的，比如日志系统、计数监控系统等。这都需要具有一定的技术功底。

最后，采用微服务架构就会涉及怎样将一个整体应用拆分成多个微服务，我们说了拆分的原则是根据业务拆分，但是这个业务边界有时候是比较难划分的。这个时候就需要有

一个超牛的架构师来做这个事儿了。

1.2 微服务中的组件与技术选型

1. 服务注册与服务发现

什么是服务注册和服务发现？为什么需要服务注册和服务发现？怎样实现服务注册和服务发现？下面就来回答这几个问题。

服务注册形象地来讲就是将服务的 ip 和 port 注册到注册中心，这里可以简单地将注册中心理解为一个 Map，其中的 key 是服务的唯一标识（可以是 serviceID，也可以是 serviceName），而 value 是一个包含 ipAndPort 的结构体的集合，例如是一个 List 集合，该 List 集合中存放了指定 service 所在的所有服务器。

服务发现简单来讲就是根据服务的唯一标识，从注册中心获取指定服务所在的服务器列表，即根据上述 Map 中的 key 来获取 value。

使用服务注册和服务发现的好处很多，其中最主要的就是实现了服务之间的解耦。如果不使用服务注册中心，那么我们需要将被调用服务的服务器列表直接写在调用服务的配置文件中，这造成了两个服务的直接耦合。之后，在调用服务的时候，通过一定的负载均衡算法，从服务器列表中获取一台服务器进行调用。此时，如果被调用服务所在服务器有宕机情况或者新添加了服务器，调用服务也不会发现，除非修改配置文件，之后还需要重启服务。而使用了服务注册和服务发现后，调用服务会及时发现新增加的机器或者宕掉的机器，而且不需要重启服务。

常用的可以用来实现注册中心的技术有 Consul、ZooKeeper、Etcd 和 Eureka 等。具体实现见第 7 章相关内容。

2. 健康检查

什么是健康检查？为什么需要健康检查？怎样实现健康检查？下面就来回答这几个问题。

健康检查是检查两个东西是否处于正常状态：一个是服务所在服务器的运行状态；一个是服务本身的运行状态。健康检查的目的其实就是为了在服务发现和服务路由的时候，可以将服务的调用请求发送到处于健康状态的机器上，不至于使服务调用因为请求被发送到不健康的机器上而失败。

常用的可以用来实现健康检查的技术有 Consul、Spring Boot 的 Actuator。具体实现见第 7 章相关内容。

3. 配置管理

为什么需要配置管理？怎样实现配置管理？下面就来回答这几个问题。

配置管理主要做三件事。第一件事是在一个地方将服务集中管理，例如在 Consul-KV 中集中配置，这样可保护配置信息的安全，例如，开发人员在上线一个项目的时候，只是将配置信息（其中的数据库配置信息是测试环境下的）提交给运维人员，运维人员可以将线上数据库的配置信息更改到配置文件中。这样的话，开发人员是看不到线上的配置信息的，起到了一定的安全防护作用。第二件事是实现服务的配置与代码分离，这样修改了配置信息之后不需要再编译、打包、部署整个服务。第三件事是实现“热配置”，即当修改了配置信息后，不需要重启服务就可以自动获取修改后的配置信息。

常用的可以用来实现配置管理的技术有 Consul、Archaius 等。具体实现见第 8 章相关内容。

4. 服务通信

什么是服务通信？为什么需要服务通信？怎样实现服务通信？下面就来回答这几个问题。

这里所说的服务通信指的是服务之间的相互调用。服务之间的调用协议可以使用 TCP 协议，也可以使用 HTTP 协议。而基于 HTTP 协议的通信方式是 Martin Fowler 所推荐的，而且基于 HTTP 协议的代码要比基于 TCP 协议的代码好写很多，因为不需要考虑丢包、粘包等比较底层的问题。当然，也可以使用现成的框架来屏蔽这些底层细节，例如 Netty。但是即使使用了 Netty，也远不如直接使用 HTTP 协议进行通信来得简单。

常用的可以用来实现服务通信的技术有 Netty、Mina、Retrofit、OkHttp 和 AsyncHttpClient 等。其中 Netty 和 Mina 出自同一人之手，风格相似，主要用于基于 TCP 或 UDP 协议的通信；OkHttp 和 AsyncHttpClient 主要用于基于 HTTP 协议的通信，后者的效率要高于前者，推荐使用后者；Retrofit 以一种接口方式封装方法，并且采用动态代理实现方法的调用，这使得调用远程方法就和调用本地方法一样简单明了。服务通信的具体实现见第 9 章相关内容。

5. 服务路由

什么是服务路由？怎样实现服务路由？下面就来回答这几个问题。

服务路由的过程是这样的：当一个请求过来时，通过服务发现和健康检查选出健康的

服务器列表，之后采用一定的负载均衡策略（路由策略）从这些服务器中选出一台，最后将请求发送到这台服务器上去。服务路由器通常会包含一个内建的负载均衡器，而且会包含多种负载均衡策略，这些策略都是可插拔的，并且会在本地缓存一份可用的服务器列表，当然，也会通过一定的策略来更新该服务器列表，例如定时地使用服务发现技术来刷新本地服务器列表缓存，进而达到在宕机或者添加了新机器的时候，本地服务器列表缓存可以及时更新。

常用的用来实现服务路由的技术有 Ribbon。

6. 服务容错

什么是服务容错？为什么需要服务容错？怎样实现服务容错？下面就来回答这几个问题。

服务容错指的是当服务集群中的一台机器宕机了，也不会导致整个服务不可用，甚至不会因为级联失败导致多个服务不可用，形成雪崩。在实际开发中，服务容错是必须要考虑的事情，不仅要考虑之前说到的级联失败的问题，还要考虑到怎样让一个宕掉的服务自动由不可用状态转为可用状态，并且让这个状态切换的时间尽可能短。

常用的可以用来实现服务容错的技术有 Hystrix。具体实现见第 10 章相关内容。

7. 日志系统

为什么需要日志系统？怎样实现日志系统？下面就来回答这几个问题。

日志系统主要用于收集散落在各台机器上的日志，并提供高效的存储和查询方式，通过清晰易懂的界面进行结果展示。当然，也会提供方便的分析功能等。

常用来实现日志系统的技术有 Logback、ELK、Redis、Flume、Hadoop、Kafka 等。具体实现见第 11 章相关内容。

8. 全链路追踪系统

什么是全链路追踪？为什么需要全链路追踪系统？怎样实现全链路追踪系统？下面就来回答这几个问题。

全链路追踪指的是在微服务架构中，由于服务比较多，通常需要多个服务彼此协作调用，这个时候就产生了调用链，我们想理清服务之间的依赖关系，可以分析调用链信息，并且最好还能以图形的形式展现出来，否则，即使统计了调用链信息，也不容易分析出依赖关系。而且如果调用链较长，想要找出耗时的服务进行调优比较难，一旦发生错误，也

很难定位是整个环节中哪一个服务发生了错误，那么通过查看调用链，我们可以尽快地找到发生问题的服务。

常用的可以用来实现链路跟踪的技术有 Zipkin、Brave 和 Spring Cloud Sleuth 等。具体实现见第 12 章相关内容。

9. 计数监控系统

为什么需要计数监控系统？怎样实现计数监控系统？下面就来回答这几个问题。

在微服务架构中，服务众多，需要对这些服务的一系列指标进行记录监控。这样，既可以根据监控数据（例如，CPU 使用率、内存占用率等）将服务调到最优，也可以让我们对自己的服务有一个实时的了解，在发生错误时，可以尽快地去处理。

常用的可以用来实现计数监控的技术有 Graphite、Grafana、Prometheus、Hystrix-Dashboard 和 Turbine 等。

10. 文档输出

什么是文档输出？为什么需要文档输出？怎样实现文档输出？下面就来回答这几个问题。

文档输出其实就是将 API 接口进行文档化，更简单地说就是通过编写代码的方式来自自动展现出 API 接口的各种信息。编写代码是开发人员的强项，如果只是添加几个注解就能将文档输出，那么这对于开发人员来讲就是信手拈来的事情。如果能将接口以一个清晰的界面展示出来，就更好了，这样开发人员就不必单独地写文档，尤其是不必刻意写一个 Word 文档。用 Word 文档输出接口信息不仅工作量巨大，而且很容易出错，接口信息一旦发生了变动，开发人员需要手动更改 Word 文档，这就有可能出错。所以使用一个良好的文档输出工具是很有必要的，不仅可以将 API 接口信息方便清晰地展现给服务调用端，而且还可以防止在一段时间后连开发人员自己都忘记了接口信息。

常用来实现文档输出的技术有 Swagger 等。具体实现见第 3 章相关内容。

11. 持续集成与持续部署系统

为什么需要持续集成与持续部署系统？怎样实现持续集成与持续部署系统？下面就来回答这几个问题。

如果没有持续集成与持续部署系统，我们需要手动管理代码，控制版本；如果没有持续集成与持续部署系统，我们需要手动将代码打成 jar 包，手动上传到一台服务器，手动关

掉之前的服务，手动启动 jar 包进程，如果是按照 Docker 镜像来部署的话，我们可能还需要手动将服务打成镜像，手动将镜像文件 push 到镜像仓库，手动将镜像文件从镜像仓库 pull 下来，手动将镜像运行起来。这一切都是手动的！效率极低，并且容易出错。所以，搭建一套持续集成与持续部署系统是刻不容缓的！

常用来实现自动化部署的技术有 GitLab、Jenkins、Docker、Kubernetes (k8s)、Mesos 及 Marathon 等。具体实现见第 13 章相关内容。

12. 服务网关

服务网关 = 路由转发 + 过滤器

路由转发指接收一切外界请求，将它们转发到后端的微服务上去；过滤器指通过类似过滤器的方式在服务网关中可以完成一系列的横切功能，例如权限校验、限流及监控等。为什么需要服务网关呢？我们看一下这个情景：我们要为我们的微服务系统添加权限校验功能，这个代码可以写在三个位置。

- 每个服务自己实现一遍。
- 写到一个公共的服务中，然后其他所有服务都依赖这个服务。
- 写到服务网关的前置过滤器中，对所有过来的请求进行权限校验。

第一种，缺点太明显，基本不用；第二种，相较于第一种好很多，代码不会有冗余，但是有两个缺点。

- 由于每个服务都引入了这个公共服务，那么相当于在每个服务中都引入了相同的权限校验的代码，使得每个服务的 jar 包大小无故增加了一些，而 jar 包，尤其是在使用 Docker 镜像进行部署的场景中，越小越好。
- 由于每个服务都引入了这个公共服务，那么后续升级这个服务可能就比较困难，而且公共服务的功能越多，升级就越难。例如，假设要改变权限校验方式，想让所有的服务都使用新的权限校验，就需要将之前所有的服务都重新打包并编译部署。

服务网关恰好可以解决这样的问题。首先，将权限校验的逻辑写在网关的过滤器中，后端服务不需要关注权限校验的代码，所以服务的 jar 包中也不会引入权限校验的逻辑，从而不会增加 jar 包大小；其次，如果想修改权限校验的逻辑，只需要修改网关中的权限校验过滤器即可，而不需要升级所有已存在的微服务。

常用来实现服务网关的技术有 Zuul、Kong 等。

13. 服务编排

服务编排主要是基于容器技术来实现一个服务的自动容错功能。例如，我们指定一个 service 有三个容器实例比较合适，如果少于三个，服务编排系统会自动创建容器实例达到三个。假设我们指定当 CPU 使用率达到 90% 时，需要再启动一个容器实例进行容错处理，服务编排系统可以自动做这个事。当下热门的服务编排系统有 Kubernetes、Mesos+Marathon 等。前者发展势头迅猛，对微服务中的概念抽象得比较好，但是还未经过大量的考验，在生产环境中使用需要慎重；后者出现较早，在生产环境中已久经考验，但是其对微服务的概念抽象得不太好。具体选型，按需而来。

第 2 章

微服务基础框架

2.1 Spring Boot 的优势

Spring 是 Java 世界中的框架之王，不管是当初风靡世界的 SSH（Struts2+Spring+Hibernate）组合，还是现在常用的 SSM（Spring MVC+Spring+MyBatis）组合，Spring 作为 IoC 容器都是必选的。Spring Boot 在 Spring 的基础上，做了更有利于开发微服务的一些工作。首先，使用 Spring Boot 后，完全做到了零配置，并且可以直接以 jar 包运行（让服务在自己的一个进程中运行），这使得部署启动非常方便；其次，starter 的引入使得 jar 包管理更加智能，我们只需要引入一个相关的 starter，Spring Boot 就会引入一系列与之相关的 jar 包，不需要我们自己来引入，这样的话，就不需要操心引入的 jar 包的版本冲突问题；最后，Spring Boot 的自动配置机制使得整合一些框架非常简单。

下面通过一个简单的入门程序来看一下 Spring Boot 的基本使用方法。

2.2 Spring Boot 入门

2.2.1 搭建项目框架

在搭建项目框架之前，先来看一下笔者使用的开发环境。

- JDK 版本: 1.8.0_73 (Spring Boot 官方推荐使用 1.8 及以上)。
- Spring Boot 版本: 1.4.3.RELEASE。
- Maven 版本: 3.3.9。
- 开发工具: IntelliJ IDEA15。

在之后的章节中, 如果没有特殊说明, 环境同上。了解了开发环境之后, 我们来搭建项目框架。这里在 IDEA 中创建了一个 Maven 项目, 项目名为 firstboot, 项目的代码结构如图 2-1 所示。

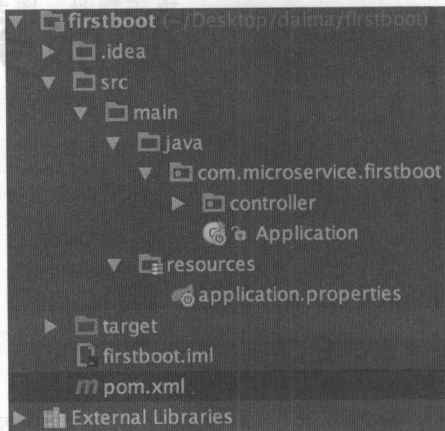


图2-1 项目的代码结构

这里为了下面介绍方便, 直接给出了最终的项目结构图, 在之后的章节中, 也会遵循这一习惯。项目框架搭建完成之后, 我们来开发第一个 hello-world 级别的 Spring Boot 应用, 通过开发该应用, 可以领略到 2.1 节中所提到的 Spring Boot 的各种优势!

2.2.2 开发第一个 Spring Boot 程序

首先来看整个项目的 pom.xml 文件, 该文件内容如下:

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://
www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.
apache.org/maven-v4_0_0.xsd">
    <modelVersion>4.0.0</modelVersion>

    <groupId>com.microservice</groupId>
```



```
<artifactId>firstboot</artifactId>
<version>1.0-SNAPSHOT</version>

<properties>
  <java.version>1.8</java.version>
</properties>
<parent>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-parent</artifactId>
  <version>1.4.3.RELEASE</version>
</parent>

<dependencies>
  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
  </dependency>
</dependencies>

<build>
  <plugins>
    <plugin>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-maven-plugin</artifactId>
    </plugin>
  </plugins>
</build>
</project>
```

对于该文件，有以下几点需要说明。

1. 在<properties>标签下通过<java.version>1.8</java.version>指定了所使用的JDK版本为1.8，这是官方推荐的方式。
2. 使用 spring-boot-starter-parent 作为项目 parent，并且指定了 Spring Boot 的版本为 1.4.3.RELEASE（这是笔者写作时的最新版本），这样 firstboot 项目就成为了一个标准的 Spring Boot 项目，这也是官方推荐的 Spring Boot 的使用方式。Spring Boot 还提供了一种不引入 starter 作为 parent 的方式，不推荐使用这种方式，这里不做介绍。
3. 引入 spring-boot-starter-web 的依赖之后，整个项目会自动引入 tomcat 和 spring-webmvc 等相关包，以支持全栈的 Web 开发。我们不需要指定该依赖的版本，

因为已经在 `spring-boot-starter-parent` 中对 `version` 进行了指定。

4. 最后，引入了 `spring-boot-maven-plugin` 插件，强烈推荐在一个 Spring Boot 项目中引入该插件，该插件会对 Maven 生成的 jar 包进行二次打包，打成一个 fat-jar 包之后，我们就可以直接使用 “`java -jar xxx.jar`” 来运行服务了，非常方便。

在创建好 `pom.xml` 文件之后，根据图 2-1，又创建了一个 `com.microservice.firstboot` 包，在该包下创建一个 `Application.java` 类。该类的代码如下：

```
package com.microservice.firstboot;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication
public class Application {
    public static void main(String[] args) {
        SpringApplication.run(Application.class, args);
    }
}
```

`Application.java` 是一个非常重要的类，通常被称为主类或启动类，是整个 Spring Boot 项目的启动入口。包含一个 `main` 方法，而且整个应用中只能有一个 `main` 方法，否则，启动会报错。在主类上需要添加注解 `@SpringBootApplication`，该注解是一个复合注解，其包含的比较好的注解是以下三个。

- `@SpringBootConfiguration`：该注解也是一个复合注解，其中最重要的注解是 `@Configuration`，指明该类由 Spring 容器管理。
- `@EnableAutoConfiguration`：该注解用于启动服务的自动配置功能。
- `@ComponentScan`：该注解用于扫描类，其作用类似于 Spring 中的 `<context:component-scan>` 标签。

细心的读者会发现一个有趣的事情：`firstboot` 项目的项目名与 `artifactId` 相同，并且在 `firstboot` 项目下有一个路径最短的包 `com.microservice.firstboot`，该包名正好是 `<groupId>.<artifactId>`，并且主类就位于该最短路径包下！注意，这不是偶然的，这是企业使用 Spring Boot 构建项目的最标准的做法。使用 `<groupId>.<artifactId>` 作为最短路径包名，不仅语义明确，而且也方便我们写 `maven-archetype`，而将主类放在最短路径包下，主要是为了方便 `@ComponentScan` 扫描整个项目中的类。

下面，我们定义了一个简单的 controller，代码如下：

```
package com.microservice.firstboot.controller;

import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestMethod;
import org.springframework.web.bind.annotation.RestController;

@RestController
@RequestMapping("/firstboot")
public class FirstBootController {

    @RequestMapping(value = "/sayHello", method = RequestMethod.GET)
    public String sayHello() {
        return "hello,this is my first boot program!!!";
    }
}
```

该类非常简单，只提供了一个接口，并返回一个 `String`。该类使用了 `@RestController` 注解，该注解是一个复合注解，其所包含的比较重要的注解是 `@Controller` 和 `@ResponseBody`，指定 controller 返回的对象自动转化为 json 格式并返回（基本类型及其包装类、`String` 除外）。

除了上述文件外，在图 2-1 中还有一个 `application.properties` 文件，该配置文件是 Spring Boot 默认读取配置信息的地方，此处不做配置。

2.2.3 运行 Spring Boot 项目

Spring Boot 程序的运行主要有以下两种方式。

1. 第一种是使用 `mvn install` 打成 jar 包，之后使用 `java -jar` 运行该 jar 包（通常在线上部署运行服务的时候使用该方式）。
2. 第二种是使用 `mvn spring-boot:run` 运行 jar 包（通常在本地 IDE 中进行调试的时候使用该方式）。

程序启动成功后，就会在控制台输出如图 2-2 所示的两行日志，从第一行可以看出 Tomcat 的启动 port，第二行是 Spring Boot 启动成功的标志。这里还给出了项目启动所花费的时间，至于该时间的由来，我们会在第 6 章做详细解析。

```
s.b.c.e.t.TomcatEmbeddedServletContainer : Tomcat started on port(s): 8080 (http)
com.microservice.firstboot.Application : Started Application in 9.462 seconds (JVM running for 10.923)
```

图2-2 程序启动后输出的日志

程序启动之后,在浏览器中输入 `http://localhost:8080/firstboot/sayHello`,输出为 `hello,this is my first boot program!!!`,则程序运行成功!!!

2.3 再学一招:使用 Maven 依赖树验证 Spring Boot 自动引包功能

在本章的“再学一招”部分,介绍一个 Maven 命令:

```
mvn dependency:tree
```

该命令在解决 jar 包冲突时很有用。当我们在 `pom.xml` 中引入了 `spring-boot-starter-web` 之后,在终端执行 `mvn dependency:tree`,依赖树如图 2-3 所示。

```
[INFO] com.microservice.firstboot:jar:1.0-SNAPSHOT
[INFO] \- org.springframework.boot:spring-boot-starter-web:jar:1.4.3.RELEASE:compile
[INFO]   +- org.springframework.boot:spring-boot-starter:jar:1.4.3.RELEASE:compile
[INFO]   | +- org.springframework.boot:spring-boot:jar:1.4.3.RELEASE:compile
[INFO]   | +- org.springframework.boot:spring-boot-autoconfigure:jar:1.4.3.RELEASE:compile
[INFO]   | +- org.springframework.boot:spring-boot-starter-logging:jar:1.4.3.RELEASE:compile
[INFO]   | | +- ch.qos.logback:logback-classic:jar:1.1.8:compile
[INFO]   | | | +- ch.qos.logback:logback-core:jar:1.1.8:compile
[INFO]   | | | \- org.slf4j:slf4j-api:jar:1.7.22:compile
[INFO]   | | +- org.slf4j:jcl-over-slf4j:jar:1.7.22:compile
[INFO]   | | +- org.slf4j:jul-to-slf4j:jar:1.7.22:compile
[INFO]   | | \- org.slf4j:log4j-over-slf4j:jar:1.7.22:compile
[INFO]   +- org.springframework:spring-core:jar:4.3.5.RELEASE:compile
[INFO]   \- org.yaml:snakeyaml:jar:1.17:runtime
[INFO] +- org.springframework.boot:spring-boot-starter-tomcat:jar:1.4.3.RELEASE:compile
[INFO]   +- org.apache.tomcat.embed:tomcat-embed-core:jar:8.5.6:compile
[INFO]   +- org.apache.tomcat.embed:tomcat-embed-el:jar:8.5.6:compile
[INFO]   \- org.apache.tomcat.embed:tomcat-embed-websocket:jar:8.5.6:compile
[INFO] +- org.hibernate:hibernate-validator:jar:5.2.4.Final:compile
[INFO]   +- javax.validation:validation-api:jar:1.1.0.Final:compile
[INFO]   +- org.jboss.logging:jboss-logging:jar:3.3.0.Final:compile
[INFO]   \- com.fasterxml.classmate:jar:1.3.3:compile
[INFO] +- com.fasterxml.jackson.core:jackson-databind:jar:2.8.5:compile
[INFO]   +- com.fasterxml.jackson.core:jackson-annotations:jar:2.8.5:compile
[INFO]   \- com.fasterxml.jackson.core:jackson-core:jar:2.8.5:compile
[INFO] +- org.springframework:spring-web:jar:4.3.5.RELEASE:compile
[INFO]   +- org.springframework:spring-aop:jar:4.3.5.RELEASE:compile
[INFO]   +- org.springframework:spring-beans:jar:4.3.5.RELEASE:compile
[INFO]   \- org.springframework:spring-context:jar:4.3.5.RELEASE:compile
[INFO] \- org.springframework:spring-webmvc:jar:4.3.5.RELEASE:compile
[INFO]   \- org.springframework:spring-expression:jar:4.3.5.RELEASE:compile
```

图2-3 依赖树

从上述依赖树可以看出，引入一个 web-starter 后，Spring Boot 自动为我们引入了下面几方面的 jar 包。

- 第一个是 spring-boot-starter，该 jar 包下包含自动配置的 jar，Logback、SLF4J 的 jar 及 spring-core。Logback 是 Spring Boot 默认使用的日志框架，我们在 Spring Boot 程序中使用它的时候，需要在 src/main/resources 下添加一个 logback.xml 文件，该文件内容如下：

```
<?xml version="1.0" encoding="UTF-8"?>
<configuration>
  <include resource="org/springframework/boot/logging/logback/base.xml"/>
</configuration>
```

- 第二个是 spring-boot-starter-tomcat，主要引入一系列 tomcat 相关的包。
- 第三个是 hibernate-validator。
- 第四个是 jackson，引入 springmvc 默认使用的 json 框架。
- 第五个是 spring-web 和 spring-webmvc。

第 3 章

微服务文档输出

3.1 Swagger 概述

Swagger 是一款可以用于设计、构建、文档化并且执行 API 的框架。使用该框架，可以轻松地创建一个 API 文档。使用 Swagger 有利于前后端分离开发，并且在测试的时候不需要再使用在浏览器中输入 URL 的方式来访问 Controller，可以直接在页面输入参数，然后单击按钮来访问。而且传统的输入 URL 的测试方式对于 post 请求的传参比较麻烦（当然，可以使用 postman 这样的浏览器插件），另外 Spring Boot 与 Swagger 的集成也非常简单。

3.2 如何使用 Swagger

3.2.1 搭建项目框架

依然使用第 2 章中的 firstboot 项目，项目的代码结构如图 3-1 所示。

在该项目中，引入了 logback.xml，用于日志记录。关于日志，我们会在第 11 章进行详细介绍。搭建好项目框架之后，可以通过代码来看一下如何在 Spring Boot 项目中使用 Swagger 生成在线的 API 文档。

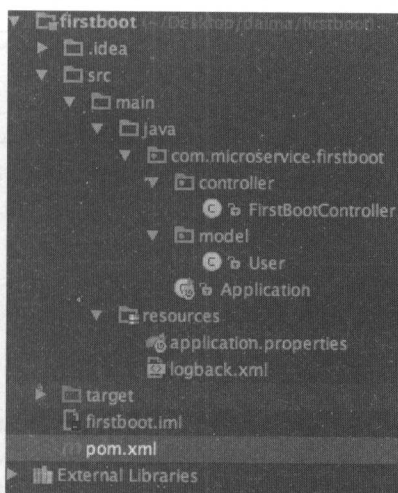


图3-1 项目的代码结构

3.2.2 Spring Boot 集成 Swagger

首先来看整个项目的 pom.xml 文件，该文件完整内容如下：

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.
org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.
apache.org/maven-v4_0_0.xsd">
    <modelVersion>4.0.0</modelVersion>

    <groupId>com.microservice</groupId>
    <artifactId>firstboot</artifactId>
    <version>1.0-SNAPSHOT</version>

    <properties>
        <java.version>1.8</java.version>
    </properties>
    <parent>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-parent</artifactId>
        <version>1.4.3.RELEASE</version>
    </parent>
```

```
<dependencies>
  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
  </dependency>
  <dependency>
    <groupId>io.springfox</groupId>
    <artifactId>springfox-swagger2</artifactId>
    <version>2.2.2</version>
  </dependency>
  <dependency>
    <groupId>io.springfox</groupId>
    <artifactId>springfox-swagger-ui</artifactId>
    <version>2.2.2</version>
  </dependency>
  <dependency>
    <groupId>org.projectlombok</groupId>
    <artifactId>lombok</artifactId>
    <version>1.16.8</version>
  </dependency>
</dependencies>

<build>
  <plugins>
    <plugin>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-maven-plugin</artifactId>
    </plugin>
  </plugins>
</build>
</project>
```

要使用 Swagger，首先需要引入 Swagger 的两个依赖：springfox-swagger2 和 springfox-swagger-ui。除了 Swagger 必需的两个依赖外，还引入了 Lombok 的依赖。Lombok 是一个主要用来消除 POJO 模板式代码（例如，getter、setter 等）的框架，当然它也可以做一些其他事情。需要注意的是，无论 Eclipse 还是 IDEA 想使用 Lombok，都要安装 Lombok 插件。关于 Lombok 的相关知识，我们会在本章的“再学一招”中介绍，此处不再赘述。

在创建好 pom.xml 文件之后，开始创建主类。主类 Application 的代码如下：

```
package com.microservice.firstboot;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import springfox.documentation.swagger2.annotations.EnableSwagger2;

@SpringBootApplication
@EnableSwagger2 //启动 Swagger
public class Application {
    public static void main(String[] args) {
        SpringApplication.run(Application.class, args);
    }
}
```

在该类中除了我们熟悉的@SpringBootApplication 注解外，还添加了@EnableSwagger2 注解，使用该注解来启动 Swagger。

为了全面地看到 Swagger 的各种使用方式，这里创建一个模型类 com.microservice.firstboot.model.User，代码如下：

```
package com.microservice.firstboot.model;

import io.swagger.annotations.ApiModel;
import io.swagger.annotations.ApiModelProperty;
import lombok.AllArgsConstructor;
import lombok.Getter;

@ApiModel("用户模型")
@AllArgsConstructor
@Getter
public class User {
    @ApiModelProperty("用户 ID")
    private int id;
    @ApiModelProperty("用户姓名")
    private String name;
    @ApiModelProperty("用户密码")
    private String password;
}
```

在该模型类中，使用了 4 个注解，分别如下。

- **@Getter**: 是一个 Lombok 注解，用来为 POJO 类生成 `getter` 方法。如果不添加该注解，我们需要为 POJO 类生成 `getter` 方法，如果一个模型类中属性较多，则整个代码中就会有大量的篇幅充斥着这种模板类代码，代码就不够简洁。
- **@AllArgsConstructor**: 是一个 Lombok 注解，用来为 POJO 类生成全参构造器。
- **@ApiModel**: 是一个 Swagger 注解，用来为一个 POJO 类做注释。
- **@ApiModelProperty**: 是一个 Swagger 注解，用来为 POJO 类中的属性做注释。

下面，创建一个简单的 `controller`，代码如下：

```
package com.microservice.firstboot.controller;

import com.microservice.firstboot.model.User;
import io.swagger.annotations.*;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestMethod;
import org.springframework.web.bind.annotation.RequestParam;
import org.springframework.web.bind.annotation.RestController;

@Api("user 相关 api")
@RestController
@RequestMapping("/user")
public class FirstBootController {

    @ApiOperation("根据 ID 获取用户信息")
    @ApiImplicitParams({
        @ApiImplicitParam(paramType="query",name="id",dataType="int",required=true,value="用户的 id",defaultValue="1")
    })
    @ApiResponses({
        @ApiResponse(code=400,message="请求参数没填好"),
        @ApiResponse(code=404,message="请求路径没有或页面跳转路径不对")
    })
    @RequestMapping(value = "/getUserInfo", method = RequestMethod.GET)
    public User getUserInfo(@RequestParam("id") int id) {
        if(id==1){
            return new User(1,"小红","123456");
        }
        return new User(2,"小刚","123456");
    }
}
```

```
}  
}
```

在该 controller 中，提供了一个简单的接口：根据 id 获取 User。该类除了使用了 Spring 的注解外，还使用了 Swagger 的 6 个注解，分别如下。

- **@Api**: 通常用来为一个 controller 类做注释，说明该 controller 的职能。
- **@ApiOperation**: 通常用来为一个接口做注释，说明该接口的职能。
- **@ApiImplicitParams**: 通常用来包含接口的一组参数注解，可以将其简单地理解为参数注解的集合。
- **@ApiImplicitParam**: 用在 **@ApiImplicitParams** 注解中，说明一个请求参数的各个方面。该注解包含的常用选项有如下。
 - **paramType**, 参数所放置的地方，包含 query、header、path、body 及 form，最常用的是前 4 个。需要注意的是，query 域中的值需要使用 **@RequestParam** 获取，header 域中的值需要使用 **@RequestHeader** 获取，path 域中的值需要使用 **@PathVariable** 获取，body 域中的值需要使用 **@RequestBody** 获取，否则可能出错。
 - **name**, 参数名。
 - **dataType**, 参数类型。
 - **required**, 参数是否必须传。
 - **value**, 参数的值。
 - **defaultValue**, 参数的默认值。
- **@ApiResponses**: 通常用来包含接口的一组响应注解，可以将其简单地理解为响应注解的集合。
- **@ApiResponse**: 用在 **@ApiResponses** 中，一般用于表达一个错误的响应信息。
 - **code**, 即 **httpCode** 数字，例如 400。
 - **message**, 信息，例如“请求参数没填好”。

这些注解就是开发中最常用的 Swagger 注解。到此为止，Spring Boot 与 Swagger 的集成就完成了。下面我们来测试 Swagger 生成的 API 文档功能。

3.2.3 分析 Swagger 生成的 API 文档

首先启动 firstboot 项目，之后，在浏览器中输入 `http://localhost:8080/swagger-ui.html`，结果如图 3-2 所示，这表示 Spring Boot 整合 Swagger 成功。注意一点，在输入 url 之后，按回车键，如果没看到结果，就多刷新几次页面，这可能是 Swagger 的一个 Bug。

user相关api : user相关api Show/Hide List Operations Expand Operations

GET /user/getUserInfo 根据ID获取用户信息

Response Class (Status 200)
Model Model Schema

用户模型 {
 id (integer, optional): 用户ID,
 name (string, optional): 用户姓名,
 password (string, optional): 用户密码
}

Response Content Type */*

Parameters

Parameter	Value	Description	Parameter Type	Data Type
id	<input type="text" value="1"/>	用户的id	query	integer

Response Messages

HTTP Status Code	Reason	Response Model	Headers
400	请求参数没填好		
401	Unauthorized		
403	Forbidden		
404	请求路径没有或页面跳转路径不对		

Try it out!

图3-2 Swagger文档

项目中的各个注解在图 3-2 中都有体现，其中 `@Api` 对应图 3-2 最上边的一行，为整个 controller 做了注释；`@ApiOperation` 对应图 3-2 上边第二行，对方法做了注释；`@ApiModel` 与 `@ApiModelProperty` 对应图 3-2 左上角的 ResponseClass，对响应类做了注释；`@ApiImplicitParam` 对应图 3-2 中的 Parameters 部分，对参数做了注释；`@ApiResponse` 对应图 3-2 中的 ResponseMessages 部分，对响应消息做了注释。

3.2.4 使用 Swagger 进行接口调用

在如图 3-2 所示的文档中，在想要运行的方法中填写好入参之后，单击“Try it out!”按钮，就可以执行该接口了！很方便，不需要再自己构造请求的 url。执行结果如图 3-3 所示。

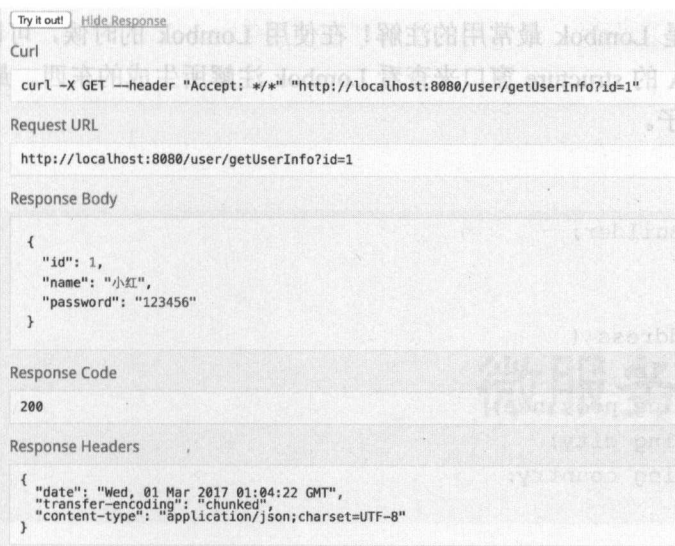


图3-3 Swagger执行结果

3.3 再学一招：使用 Lombok 消除 POJO 类模板代码

在本章的“再学一招”部分，介绍一下 Lombok 的常见用法及注意事项。

首先，要让 IDE 支持 Lombok。对于 IDEA 来说，安装 Lombok 插件，只需要在 preferences->plugins 中搜索 lombok plugin 并安装即可；对于 Eclipse 而言，先下载 Lombok.jar (<https://projectlombok.org/download.html>)，然后双击 lombok.jar，之后一路回车，直到选中 eclipse.ini 文件，单击 install/update 按钮，最后重启 Eclipse 即可。

Lombok 常用注解如下。

- **@Getter**: 用于生成 getter 方法，可用在类或属性上。
- **@Setter**: 用于生成 setter 方法，可用在类或属性上。
- **@AllArgsConstructor**: 用于生成全参构造器，用在类上。
- **@NoArgsConstructor**: 用于生成无参构造器，用在类上。
- **@Builder**: 用于将类改造成 builder 模式，用在类、方法或构造器上。
- **@Data**: 是一个复合注解，使用该注解，会生成默认的无参构造器、所有属性的 getter、所有非 final 的属性的 setter 方法，重写 toString 方法，重写 equals 方法，重写 hashCode 方法。

3.2 这些注解就是 Lombok 最常用的注解！在使用 Lombok 的时候，可以配合 Eclipse 的 outline 或者 IDEA 的 structure 窗口来查看 Lombok 注解所生成的东西。最后，简单地看一个 @Builder 的例子。

1. Address

```
import lombok.Builder;

@Builder
public class Address {
    private int id;
    private String province;
    private String city;
    private String country;
}
```

当我们在 Address 类上使用了 @Builder 注解后，就可以使用如下的方式来创建对象并初始化。

2. 使用方法

```
Address address = Address.builder().province("内蒙古自治区")
    .city("呼和浩特市")
    .country("回民区")
    .build();
```

第4章

微服务数据库

4.1 单数据源

在微服务中，通常一个服务只会使用一个数据库，所以我们首先以最简单的单数据源为例来介绍在微服务中怎样操作数据库。

4.1.1 搭建项目框架

在 IDEA 中创建一个 Maven 项目，项目名称为 dbandcache，项目的代码结构如图 4-1 所示。

搭建好项目框架之后，进行数据库和数据表的创建。笔者使用的是 MySQL，所以需要先在机器上安装 MySQL。安装过程此处不再赘述，笔者使用的 MySQL 版本是 5.7。为了方便，笔者还推荐使用一款 MySQL 的可视化客户端：Navicat，通过 Navicat 可以方便地进行数据库的操作。

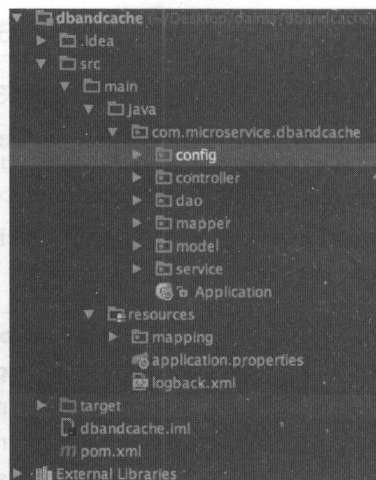


图4-1 项目的代码结构

4.1.2 建库和建表

首先，创建数据库，库名为 `microservicedb1`，之后执行如下的 SQL 语句建表 `t_user`，并插入一条测试数据。

```
SET NAMES utf8;
SET FOREIGN_KEY_CHECKS = 0;

DROP TABLE IF EXISTS `t_user`;
CREATE TABLE `t_user` (
  `id` bigint(20) NOT NULL AUTO_INCREMENT,
  `name` varchar(50) NOT NULL COMMENT '用户名',
  `phone` varchar(30) NOT NULL COMMENT '手机',
  PRIMARY KEY (`id`)
) ENGINE=InnoDB AUTO_INCREMENT=2 DEFAULT CHARSET=utf8;

BEGIN;
INSERT INTO `t_user` VALUES ('1', '小娜', '15088887777');
COMMIT;

SET FOREIGN_KEY_CHECKS = 1;
```

建好表之后，就可以开始项目的开发了。

4.1.3 使用 MyBatis-Generator 生成数据访问层

这里使用 `MyBatis-Generator` 来生成数据访问层的模板代码。关于 `MyBatis-Generator`，我们会在本章的“再学一招：MyBatis-Generator 基本用法”部分进行详细介绍。

首先，下载 `mybatis-generator-core-x.x.x.jar` 和 `mysql-connector-java-x.x.x.jar`，并且创建 `MyBatis-Generator` 的配置文件 `generatorConfig-user.xml`。最后将这些文件组成如图 4-2 所示的目录结构。

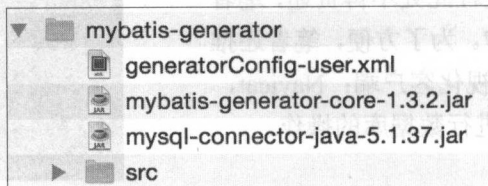


图4-2 MyBatis-Generator目录结构

之后, 填写 generatorConfig-user.xml 的内容如下:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE generatorConfiguration
PUBLIC "-//mybatis.org//DTD MyBatis Generator Configuration 1.0//EN"
"http://mybatis.org/dtd/mybatis-generator-config_1_0.dtd">
<generatorConfiguration>
  <!--数据库驱动-->
  <classPathEntry location="mysql-connector-java-5.1.37.jar"/>
  <context id="mysql">
    <commentGenerator>
      <property name="suppressAllComments" value="true"/>
    </commentGenerator>
    <!--数据库链接地址账号密码-->
    <jdbcConnection driverClass="com.mysql.jdbc.Driver" connectionURL=
"jdbc:mysql://127.0.0.1:3306/microservicedb1" userId="root" password="123456">
    </jdbcConnection>
    <!--生成 Model 类存放位置-->
    <javaModelGenerator targetPackage="com.microservice.dbandcache.model"
targetProject="src">
      <property name="enableSubPackages" value="true"/>
      <property name="trimStrings" value="true"/>
    </javaModelGenerator>
    <!--生成映射文件存放位置-->
    <sqlMapGenerator targetPackage="lcw.mapping" targetProject="src">
      <property name="enableSubPackages" value="true"/>
    </sqlMapGenerator>
    <!--生成 mapper 接口存放位置-->
    <javaClientGenerator type="XMLMAPPER" targetPackage="com.microservice.
dbandcache.mapper" targetProject="src">
      <property name="enableSubPackages" value="true"/>
    </javaClientGenerator>
    <!--对应表及生成类名-->
    <table tableName="t_user" domainObjectName="User"
enableCountByExample="false" enableUpdateByExample="false"
enableDeleteByExample="false" enableSelectByExample="false"
selectByExampleQueryId="false">
    </table>
  </context>
</generatorConfiguration>
```

在该文件中指定了数据库的连接四要素（driver、connectionurl、username、password），生成的 model 类、mapper 接口的位置以及映射文件的存储目录，最后指定了哪一张表（tableName）对应哪一个 model（domainObjectName）。这里值得注意的是，model 类和 mapper 类的位置一定要和如图 4-1 所示的项目代码结构相对应（即包名要写对），否则在代码生成之后，还需要进行修改。

之后，从终端进入 MyBatis-Generator 目录下，执行如下语句：

```
java -jar mybatis-generator-core-1.3.2.jar -configfile generatorConfig-user.xml -  
overwrite
```

之后，在 src 目录下我们会看到生成的 User 类、UserMapper 接口及 UserMapper.xml 文件。将这些文件复制到相应的目录下即可。如果在执行语句时出现错误，则需要先创建 src 文件夹。

4.1.4 Spring Boot 集成 MyBatis

MyBatis 作为一个半自动的 ORM 框架，由于其极大的灵活性以及高效的性能，越来越受到广大程序的欢迎。在本节将会使用 Spring Boot 来集成 MyBatis，之后通过 MyBatis 来操作数据库。

先来看一下 pom.xml 文件的内容：

```
<?xml version="1.0" encoding="UTF-8"?>  
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.  
org/2001/XMLSchema-instance"  
    xsi:schemaLocation="http://maven.apache.org/POM/4.0.0  
http://maven.apache.org/maven-v4_0_0.xsd">  
    <modelVersion>4.0.0</modelVersion>  
  
    <groupId>com.microservice</groupId>  
    <artifactId>dbandcache</artifactId>  
    <version>1.0-SNAPSHOT</version>  
  
    <properties>  
        <java.version>1.8</java.version>  
    </properties>  
    <parent>  
        <groupId>org.springframework.boot</groupId>
```

```
<artifactId>spring-boot-starter-parent</artifactId>
<version>1.4.3.RELEASE</version>
</parent>

<dependencies>
  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
  </dependency>
  <dependency>
    <groupId>io.springfox</groupId>
    <artifactId>springfox-swagger2</artifactId>
    <version>2.2.2</version>
  </dependency>
  <dependency>
    <groupId>io.springfox</groupId>
    <artifactId>springfox-swagger-ui</artifactId>
    <version>2.2.2</version>
  </dependency>
  <dependency>
    <groupId>org.projectlombok</groupId>
    <artifactId>lombok</artifactId>
    <version>1.16.8</version>
  </dependency>
  <!-- 与数据库操作相关的依赖 -->
  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-jdbc</artifactId>
  </dependency>
  <!-- 数据源 -->
  <dependency>
    <groupId>com.alibaba</groupId>
    <artifactId>druid</artifactId>
    <version>1.0.14</version>
  </dependency>
  <!-- mysql -->
  <dependency>
    <groupId>mysql</groupId>
    <artifactId>mysql-connector-java</artifactId>
    <scope>runtime</scope>
```



```

</dependency>
<!-- mybatis -->
<dependency>
    <groupId>org.mybatis</groupId>
    <artifactId>mybatis</artifactId>
    <version>3.2.8</version>
</dependency>
<dependency>
    <groupId>org.mybatis</groupId>
    <artifactId>mybatis-spring</artifactId>
    <version>1.2.2</version>
</dependency>
</dependencies>

<build>
    <plugins>
        <plugin>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-maven-plugin</artifactId>
        </plugin>
    </plugins>
</build>
</project>

```

在该文件中引入了与数据库相关的 5 个包：spring-boot-starter-jdbc、druid、mysql-connector-java、mybatis 及 mybatis-spring。其中，mysql-connector-java 的 scope 是 runtime，使用的数据源是阿里巴巴的 Druid。

然后看一下启动主类 com.microservice.dbandcache.Application 的代码：

```

package com.microservice.dbandcache;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import springfox.documentation.swagger2.annotations.EnableSwagger2;

@SpringBootApplication
@EnableSwagger2
public class Application {
    public static void main(String[] args) {

```

```
SpringApplication.run(Application.class, args);  
}  
}
```

在 4.1.3 节中，生成了三个数据访问层的模板文件，下面分别来看一下。首先是模型类 `com.microservice.dbandcache.model.User`，代码如下：

```
package com.microservice.dbandcache.model;  
  
import lombok.Getter;  
import lombok.Setter;  
  
@Getter  
@Setter  
public class User {  
    private Long id;  
    private String name;  
    private String phone;  
}
```

该类是由 MyBatis-Generator 生成的，这里将生成的 User 类的 getter 和 setter 使用 Lombok 的注解来替代了，使代码简洁一些。

然后是 mapper 接口 `com.microservice.dbandcache.mapper.UserMapper`，代码如下：

```
package com.microservice.dbandcache.mapper;  
  
import com.microservice.dbandcache.model.User;  
  
public interface UserMapper {  
    int deleteByPrimaryKey(Long id);  
    int insert(User record);  
    int insertSelective(User record);  
    User selectByPrimaryKey(Long id);  
    int updateByPrimaryKeySelective(User record);  
    int updateByPrimaryKey(User record);  
}
```

该类是由 MyBatis-Generator 生成的第二个模板文件，其提供了最基本的 6 个接口。这里生成的 6 个方法声明以及在 `UserMapper.xml` 中生成的对应的 6 个 SQL，我们暂时不要删

掉,以备以后使用,或者等整个项目已经完成了,到了测试阶段再去删掉没用的方法和 SQL。这里暂时不删,但是为了减少篇幅,在之后的代码中,会删掉由其他数据表生成的无用的模板代码。

MyBatis-Generator 生成的第三个模板文件是 `src/main/resources/mapping/UserMapper.xml`, 代码如下:

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE mapper PUBLIC "-//mybatis.org/DTD Mapper 3.0//EN" "http://mybatis.org/dtd/mybatis-3-mapper.dtd" >
<mapper namespace="com.microservice.dbandcache.mapper.UserMapper">
    <resultMap id="BaseResultMap" type="com.microservice.dbandcache.model.User">
        <id column="id" property="id" jdbcType="BIGINT"/>
        <result column="name" property="name" jdbcType="VARCHAR"/>
        <result column="phone" property="phone" jdbcType="VARCHAR"/>
    </resultMap>
    <sql id="Base_Column_List">
        id, name, phone
    </sql>
    <select id="selectByPrimaryKey" resultMap="BaseResultMap" parameterType="java.lang.Long">
        select
        <include refid="Base_Column_List"/>
        from t_user
        where id = #{id,jdbcType=BIGINT}
    </select>
    <delete id="deleteByPrimaryKey" parameterType="java.lang.Long">
        delete from t_user
        where id = #{id,jdbcType=BIGINT}
    </delete>
    <insert id="insert" parameterType="com.microservice.dbandcache.model.User">
        insert into t_user (id, name, phone)
        values (#{id,jdbcType=BIGINT}, #{name,jdbcType=VARCHAR}, #{phone,
jdbcType=VARCHAR})
    </insert>
    <insert id="insertSelective"
parameterType="com.microservice.dbandcache.model.User">
        insert into t_user
        <trim prefix="(" suffix=")" suffixOverrides=",">
            <if test="id != null">
```



```

        id,
    </if>
    <if test="name != null">
        name,
    </if>
    <if test="phone != null">
        phone,
    </if>
</trim>
<trim prefix="values (" suffix=")" suffixOverrides=",">
    <if test="id != null">
        #{id,jdbcType=BIGINT},
    </if>
    <if test="name != null">
        #{name,jdbcType=VARCHAR},
    </if>
    <if test="phone != null">
        #{phone,jdbcType=VARCHAR},
    </if>
</trim>
</insert>
<update id="updateByPrimaryKeySelective" parameterType="com.microservice.
dbandcache.model.User">
    update t_user
    <set>
        <if test="name != null">
            name = #{name,jdbcType=VARCHAR},
        </if>
        <if test="phone != null">
            phone = #{phone,jdbcType=VARCHAR},
        </if>
    </set>
    where id = #{id,jdbcType=BIGINT}
</update>
<update id="updateByPrimaryKey" parameterType="com.microservice.
dbandcache.model.User">
    update t_user
    set name = #{name,jdbcType=VARCHAR}, phone = #{phone,jdbcType=VARCHAR}
    where id = #{id,jdbcType=BIGINT}
</update>

```

```
</mapper>
```

该 xml 文件生成了 6 个最基本的 SQL，包括最基本的增、删、改、查，条件不定式插入与更新。

接下来在 `src/main/resources/application.properties` 文件中配置数据库信息，配置如下：

```
microservicedb1.jdbc.driverClassName = com.mysql.jdbc.Driver
microservicedb1.jdbc.url =
jdbc:mysql://localhost:3306/microservicedb1?zeroDateTimeBehavior=convertToNull&useUnicode=true&characterEncoding=utf-8
microservicedb1.jdbc.username = root
microservicedb1.jdbc.password = 123456
```

这里定义了连接数据库的四要素。

一切准备就绪之后，开始集成 MyBatis，代码如 `com.microservice.dbandcache.config.MyBatisConfig` 类所示：

```
package com.microservice.dbandcache.config;

import java.util.Properties;
import javax.sql.DataSource;
import org.apache.ibatis.session.SqlSessionFactory;
import org.mybatis.spring.SqlSessionFactoryBean;
import org.mybatis.spring.annotation.MapperScan;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.core.env.Environment;
import org.springframework.core.io.support.PathMatchingResourcePatternResolver;
import com.alibaba.druid.pool.DruidDataSourceFactory;

@Configuration
@MapperScan(basePackages = "com.microservice.dbandcache.mapper")
public class MyBatisConfig {
    @Autowired
    private Environment env;

    @Bean
```

```

public DataSource dataSource() throws Exception {
    Properties props = new Properties();
    props.put("driverClassName",
env.getProperty("microservicedbl.jdbc.driverClassName"));
    props.put("url", env.getProperty("microservicedbl.jdbc.url"));
    props.put("username",
env.getProperty("microservicedbl.jdbc.username"));
    props.put("password",
env.getProperty("microservicedbl.jdbc.password"));
    return DruidDataSourceFactory.createDataSource(props);
}

@Bean
public SqlSessionFactory sqlSessionFactory(DataSource ds) throws Exception
{
    SqlSessionFactoryBean fb = new SqlSessionFactoryBean();
    fb.setDataSource(ds); //指定数据源
    fb.setTypeAliasesPackage("com.microservice.dbandcache.model"); //指定基包
    fb.setMapperLocations(new PathMatchingResourcePatternResolver().
getResources("classpath:mapping/*.xml")); //指定 xml 文件位置
    return fb.getObject();
}
}

```

这个类是整个项目中最重要类。其中，`@MapperScan` 注解用来指定扫描的 `mapper` 接口所在的包，读取配置文件的是 `org.springframework.core.env.Environment` 实例。整个类的流程为：首先根据数据库配置创建一个 `DataSource` 单例，之后根据该 `DataSource` 实例和 `SqlSessionFactoryBean` 创建一个 `SqlSessionFactory` 单例。之后 `SqlSessionFactory` 创建出 `SqlSession`，再使用 `SqlSession` 获取相应的 `Mapper` 实例，然后通过 `Mapper` 实例就可以肆意地操作数据库了。`SqlSessionFactoryBean` 中的 `TypeAliasesPackage` 用来指定 `domain` 类的基包，即指定其在 `xxxMapper.xml` 文件中可以使用简名来代替全类名；`MapperLocations` 用来指定 `xxxMapper.xml` 文件所在的位置，如果 `MyBatis` 完全使用注解，则也可以不设置这两个参数。值得注意的是，这里使用的数据源是阿里巴巴的 `Druid`，而 `Spring Boot` 默认使用的是 `tomcat-jdbc` 数据源。

`Spring Boot` 与 `MyBatis` 集成之后，遵循最基本的分层架构。笔者设计了三个类：一个 `dao`、一个 `service` 和一个 `controller`。

首先是 `com.microservice.dbandcache.dao.UserDao`，代码如下：

```
package com.microservice.dbandcache.dao;

import com.microservice.dbandcache.mapper.UserMapper;
import com.microservice.dbandcache.model.User;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Repository;

@Repository
public class UserDao {
    @Autowired
    private UserMapper userMapper;

    public User selectByPrimaryKey(long id) {
        return userMapper.selectByPrimaryKey(id);
    }
}
```

注入 `UserMapper`，调用 `selectByPrimaryKey(long id)` 方法。

然后是 `com.microservice.dbandcache.service.UserService`，代码如下：

```
package com.microservice.dbandcache.service;

import com.microservice.dbandcache.dao.UserDao;
import com.microservice.dbandcache.model.User;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;

@Service
public class UserService {
    @Autowired
    private UserDao userDao;

    public User getUser(long id) {
        return userDao.selectByPrimaryKey(id);
    }
}
```

注入 `UserDao`，调用 `selectByPrimaryKey(long id)` 方法。

最后是 `com.microservice.dbandcache.controller.DbAndCacheController`，代码如下：

```
package com.microservice.dbandcache.controller;

import com.microservice.dbandcache.model.User;
import com.microservice.dbandcache.service.UserService;
import io.swagger.annotations.*;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestMethod;
import org.springframework.web.bind.annotation.RequestParam;
import org.springframework.web.bind.annotation.RestController;

@Api("user 相关 api")
@RestController
@RequestMapping("/user")
public class DbAndCacheController {
    @Autowired
    private UserService userService;

    @ApiOperation("根据 ID 获取用户信息")
    @ApiImplicitParams({
        @ApiImplicitParam(paramType="query", name="id", dataType="long", required=true, value="用户的 id", defaultValue="1")
    })
    @ApiResponses({
        @ApiResponse(code=401, message="权限校验不通过")
    })
    @RequestMapping(value = "/getUserInfo", method = RequestMethod.GET)
    public User getUserInfo(@RequestParam("id") long id) {
        return userService.getUser(id);
    }
}
```

注入 `UserService`，调用其 `getUser(long id)` 方法。整个项目的功能很简单，即根据用户 ID 从数据库获取用户信息。完成了代码后，运行程序，使用 `Swagger` 进行测试。

4.2 多数据源

对于大部分情况，一个服务只需要使用一个数据源，但是有的时候，服务内部逻辑比

较复杂，也会需要访问多个数据源（常见的读写分离，其实就是对两个数据源进行操作）。

假设我们现在有两个数据源 `microservicedb1` 和 `microservicedb2`，用户信息表 `t_user` 存在 `microservicedb1` 中，用户车的信息表 `t_car` 存在 `microservicedb2` 中。`dbandcache` 这个服务根据用户 ID 查询出用户信息的同时也要查询出该用户车的信息，一起返回给前端。这个时候，就出现了一个服务需要访问两个数据源的情况。下面我们开始完成这个需求。

4.2.1 建库和建表

首先需要创建库和表，并初始化数据。`microservicedb1` 及其中的 `t_user` 表在 4.1.2 节中已经创建好了，接下来创建库 `microservicedb2`，之后在其中执行如下的 SQL 语句：

```
SET NAMES utf8;
SET FOREIGN_KEY_CHECKS = 0;

DROP TABLE IF EXISTS `t_car`;
CREATE TABLE `t_car` (
  `id` bigint(20) NOT NULL AUTO_INCREMENT,
  `name` varchar(30) NOT NULL COMMENT '车名',
  `color` varchar(255) NOT NULL COMMENT '车的颜色',
  `owner` bigint(20) NOT NULL COMMENT '车主',
  PRIMARY KEY (`id`),
  KEY `idx_owner` (`owner`) USING BTREE
) ENGINE=InnoDB AUTO_INCREMENT=2 DEFAULT CHARSET=utf8;

BEGIN;
INSERT INTO `t_car` VALUES ('1', '奥迪', 'red', '1');
COMMIT;

SET FOREIGN_KEY_CHECKS = 1;
```

该表中有一个字段 `owner`，其对应 `t_user` 表中的 `id`，可以将其理解为逻辑外键。

建好表并初始化好数据之后，使用 `MyBatis-Generator` 来生成 `car` 相关的 `model` 类、`mapper` 接口及 `xml` 文件，之后将这些类复制到 `dbandcache` 服务中，再新增加一些类，并修改一些类即可。

4.2.2 使用 MyBatis-Generator 生成数据访问层

MyBatis-Generator 生成的 `com.microservice.dbandcache.model.Car` 代码如下：

```
package com.microservice.dbandcache.model;

import lombok.Getter;
import lombok.Setter;

@Getter
@Setter
public class Car {
    private Long id;
    private String name;
    private String color;
    private Long owner;
}
```

MyBatis-Generator 生成的 `com.microservice.dbandcache.mapper.CarMapper` 代码如下：

```
package com.microservice.dbandcache.mapper;

import com.microservice.dbandcache.model.Car;
import org.apache.ibatis.annotations.Param;

import java.util.List;

public interface CarMapper {
    List<Car> selectByOwner(@Param("ownerId") long ownerId);
}
```

这里将多余的不使用的方法删掉了。

MyBatis-Generator 生成的 `src/main/resources/mapping/CarMapper.xml` 代码如下：

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE mapper PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN"
"http://mybatis.org/dtd/mybatis-3-mapper.dtd" >
<mapper namespace="com.microservice.dbandcache.mapper.CarMapper">
    <resultMap id="BaseResultMap"
type="com.microservice.dbandcache.model.Car">
```

```

        <id column="id" property="id" jdbcType="BIGINT"/>
        <result column="name" property="name" jdbcType="VARCHAR"/>
        <result column="color" property="color" jdbcType="VARCHAR"/>
        <result column="owner" property="owner" jdbcType="BIGINT"/>
    </resultMap>
    <sql id="Base_Column_List">
        id, name, color, owner
    </sql>
    <select id="selectByOwner" resultMap="BaseResultMap"
parameterType="java.lang.Long">
        select
        <include refid="Base_Column_List"/>
        from t_car
        where owner = #{ownerId,jdbcType=BIGINT}
    </select>
</mapper>

```

这里依然将多余的不使用的方法删掉了。

之后配置数据库信息，src/main/resources/application.properties 配置如下：

```

microservicedb1.jdbc.driverClassName = com.mysql.jdbc.Driver
microservicedb1.jdbc.url =
jdbc:mysql://localhost:3306/microservicedb1?zeroDateTimeBehavior=convertToNu
ll&amp;useUnicode=true&amp;characterEncoding=utf-8
microservicedb1.jdbc.username = root
microservicedb1.jdbc.password = 123456

microservicedb2.jdbc.driverClassName = com.mysql.jdbc.Driver
microservicedb2.jdbc.url =
jdbc:mysql://localhost:3306/microservicedb2?zeroDateTimeBehavior=convertToNu
ll&amp;useUnicode=true&amp;characterEncoding=utf-8
microservicedb2.jdbc.username = root
microservicedb2.jdbc.password = 123456

```

在该配置文件中新增了 microservicedb2 数据库的配置。做好准备工作之后，开始实现多数据源！

4.2.3 结合 AbstractRoutingDataSource 实现动态数据源

首先定义一个枚举类，com.microservice.dbandcache.config.DatabaseType 代码如下：

```
package com.microservice.dbandcache.config;
```

```
public enum DatabaseType {  
    microservicedb1,microservicedb2  
}
```

该类列出所有的数据源 key。之后创建一个数据源 key 持有类 `com.microservice.dbandcache.config.DatabaseContextHolder`，代码如下：

```
package com.microservice.dbandcache.config;
```

```
public class DatabaseContextHolder {  
    private static final ThreadLocal<DatabaseType> contextHolder = new  
    ThreadLocal<>();  
  
    public static void setDatabaseType(DatabaseType type) {  
        contextHolder.set(type);  
    }  
  
    public static DatabaseType getDatabaseType() {  
        return contextHolder.get();  
    }  
}
```

该类主要用于在选择数据源时，将相应的数据源的 key 设置到 contextHolder 中，之后对数据库的访问，就使用该 key 对应的数据源。

然后继承 `springjdbc` 的 `AbstractRoutingDataSource`，实现动态数据源 `com.microservice.dbandcache.config.DynamicDataSource`，代码如下：

```
package com.microservice.dbandcache.config;
```

```
import org.springframework.jdbc.datasource.lookup.AbstractRoutingDataSource;  
  
public class DynamicDataSource extends AbstractRoutingDataSource {  
    protected Object determineCurrentLookupKey() {  
        return DatabaseContextHolder.getDatabaseType();  
    }  
}
```

该类实现了继承自 `AbstractRoutingDataSource` 的 `determineCurrentLookupKey()` 方法, 在该方法中通过调用数据源 `key` 持有类 `DatabaseContextHolder` 的 `getDatabaseType()` 方法来获取数据源 `key`。

然后, 在 `com.microservice.dbandcache.config.MyBatisConfig` 类中, 构造动态数据源, 集成 MyBatis, 代码如下:

```
package com.microservice.dbandcache.config;

import java.util.HashMap;
import java.util.Map;
import java.util.Properties;
import javax.sql.DataSource;
import org.apache.ibatis.session.SqlSessionFactory;
import org.mybatis.spring.SqlSessionFactoryBean;
import org.mybatis.spring.annotation.MapperScan;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.beans.factory.annotation.Qualifier;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.context.annotation.Primary;
import org.springframework.core.env.Environment;
import org.springframework.core.io.support.PathMatchingResourcePatternResolver;
import com.alibaba.druid.pool.DruidDataSourceFactory;

@Configuration
@MapperScan(basePackages = "com.microservice.dbandcache.mapper")
public class MyBatisConfig {
    @Autowired
    private Environment env;

    @Bean
    public DataSource microservicedblDataSource() throws Exception {
        Properties props = new Properties();
        props.put("driverClassName",
env.getProperty("microservicedbl.jdbc.driverClassName"));
        props.put("url", env.getProperty("microservicedbl.jdbc.url"));
        props.put("username",
```



```

env.getProperty("microservicedb1.jdbc.username"));
    props.put("password",
env.getProperty("microservicedb1.jdbc.password"));
    return DruidDataSourceFactory.createDataSource(props);
}

@Bean
public DataSource microservicedb2DataSource() throws Exception {
    Properties props = new Properties();
    props.put("driverClassName",
env.getProperty("microservicedb2.jdbc.driverClassName"));
    props.put("url", env.getProperty("microservicedb2.jdbc.url"));
    props.put("username",
env.getProperty("microservicedb2.jdbc.username"));
    props.put("password",
env.getProperty("microservicedb2.jdbc.password"));
    return DruidDataSourceFactory.createDataSource(props);
}

@Bean
@Primary
public DynamicDataSource
dataSource(@Qualifier("microservicedb1DataSource") DataSource
microservicedb1DataSource,
@Qualifier("microservicedb2DataSource")
DataSource microservicedb2DataSource) {
    Map<Object, Object> targetDataSources = new HashMap<>();
    targetDataSources.put(DatabaseType.microservicedb1,
microservicedb1DataSource);
    targetDataSources.put(DatabaseType.microservicedb2,
microservicedb2DataSource);

    DynamicDataSource dataSource = new DynamicDataSource();
    dataSource.setTargetDataSources(targetDataSources);
// 该方法是 AbstractRoutingDataSource 的方法
    dataSource.setDefaultTargetDataSource(microservicedb1DataSource);
// 默认的 datasource 设置为 myTestDbDataSource

    return dataSource;
}

```

```

@Bean
public SqlSessionFactory sqlSessionFactory(DynamicDataSource ds) throws
Exception {
    SqlSessionFactoryBean fb = new SqlSessionFactoryBean();
    fb.setDataSource(ds); //指定数据源
    fb.setTypeAliasesPackage("com.microservice.dbandcache.model"); //指定基包
    fb.setMapperLocations(new PathMatchingResourcePatternResolver().
getResources("classpath:mapping/*.xml")); //指定 xml 文件位置
    return fb.getObject();
}
}

```

在该类中首先创建了两个数据源 `microservicedb1DataSource` 和 `microservicedb2DataSource`，之后将这两个数据源设置到 `DynamicDataSource` 数据源中。在 `DynamicDataSource` 中设置了目标数据源 `map`，并且设置了默认的数据源为 `microservicedb1DataSource`，这样以后就不需要为访问 `microservicedb1DataSource` 的 `dao` 类选择数据源了，直接使用默认的数据源。也就是说我们不需要显式为 `UserDao` 选择数据源，会默认选择 `microservicedb1DataSource`。而对于下边的 `CarDao`，就需要显式指定其访问的数据源为 `microservicedb2DataSource`。另外，值得注意的是，`MyBatisConfig` 中的三个数据源都是 `javax.sql.DataSource` 接口的实现或实现的子类，所以在 `DynamicDataSource` 类上添加了 `@Primary` 注解，该注解的作用是“指定在同一个接口有多个实现类可以注入的时候，默认选择哪一个，而不是让 Spring 因为多个注入选择而不知道该选哪个最终导致报错”。

然后依然根据分层规则，编写 `dao`、`service` 和 `controller` 类。

`com.microservice.dbandcache.dao.CarDao` 代码如下：

```

package com.microservice.dbandcache.dao;

import com.microservice.dbandcache.config.DatabaseContextHolder;
import com.microservice.dbandcache.config.DatabaseType;
import com.microservice.dbandcache.mapper.CarMapper;
import com.microservice.dbandcache.model.Car;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Repository;

import com.microservice.dbandcache.mapper.UserMapper;

```



```
import com.microservice.dbandcache.model.User;

import java.util.List;

@Repository
public class CarDao {
    @Autowired
    private CarMapper carMapper;

    public List<Car> selectByOwner(long ownerId) {
        DatabaseContextHolder.setDatabaseType(DatabaseType.microservicedb2);
        return carMapper.selectByOwner(ownerId);
    }
}
```

说明：该类在调用 mapper 操作数据库之前，首先使用 DatabaseContextHolder.setDatabaseType(DatabaseType.microservicedb2)选择数据源，之后再进行数据库操作。

com.microservice.dbandcache.service.UserService 代码如下：

```
public UserAndCar getUserAndCar(long userId) {
    UserAndCar userAndCar = new UserAndCar();
    User user = userDao.selectByPrimaryKey(userId);
    if (user != null) {
        List<Car> cars = carDao.selectByOwner(userId);
        userAndCar.setId(user.getId());
        userAndCar.setName(user.getName());
        userAndCar.setPhone(user.getPhone());
        userAndCar.setCars(cars);
    }
    return userAndCar;
}
```

这里只列出关键代码。其中，UserAndCar 类代码如下：

```
package com.microservice.dbandcache.model;

import lombok.Getter;
import lombok.NoArgsConstructor;
import lombok.Setter;
```

```
import java.util.List;

@Getter
@Setter
@NoArgsConstructor
public class UserAndCar extends User {
    private List<Car> cars;
}
```

最后看一下 `com.microservice.dbandcache.controller.DbAndCacheController` 的代码：

```
@ApiOperation("根据用户 ID 获取用户及其车辆信息")
@ApiImplicitParams({
    @ApiImplicitParam(paramType="query",name="id",dataType="long",
        required=true,value="用户的 id",defaultValue="1")
})
@RequestMapping(value = "/getUserAndCar", method = RequestMethod.GET)
public UserAndCar getUserAndCar(@RequestParam("id") long id) {
    return userService.getUserAndCar(id);
}
```

至此，我们就实现了微服务中对多数据源的使用。然后运行程序，使用 Swagger 进行测试即可。但是如果一个服务中的 dao 类比较多，那么可能需要写很多遍如下的代码：`DatabaseContextHolder.setDatabaseType(DatabaseType.microservicedb2);`。有没有什么好办法，让所有的 dao 类还是按照之前一个数据源时的写法，即不要在每个 dao 类中都写上这句代码呢？答案是肯定的，使用 AOP 来实现。

4.2.4 使用 AOP 简化数据源选择功能

编写一个切面类 `com.microservice.dbandcache.config.DataSourceAspect`，代码如下：

```
package com.microservice.dbandcache.config;

import com.microservice.dbandcache.dao.CarDao;
import org.aspectj.lang.JoinPoint;
import org.aspectj.lang.annotation.Aspect;
import org.aspectj.lang.annotation.Before;
import org.springframework.stereotype.Component;
```

```

@Aspect
@Component
public class DataSourceAspect {
    @Before("execution(* com.microservice.dbandcache.dao.*.*(..))")
    public void setDataSourceKey(JoinPoint point) {
        if (point.getTarget().instanceof CarDao) {
            DatabaseContextHolder.setDatabaseType(DatabaseType.microservicedb2);
        }
    }
}

```

该类拦截 `com.microservice.dbandcache.dao` 包下的所有类的所有方法（不管返回值和入参是什么），将其中的 `CarDao` 类的数据源设置为 `microservicedb2`。这样，`CarDao` 中的

`DatabaseContextHolder.setDatabaseType(DatabaseType.microservicedb2);` 就可以删掉了。

到此为止，Spring Boot 集成 MyBatis 来实现多数据源就大功告成了！

4.2.5 实现多数据源的步骤总结

最后简要总结一下用以上方法实现多数据源的步骤。

1. `DatabaseType` 列出所有的数据源的 key 作为第 5 步中所说的 key。
2. `DatabaseContextHolder` 是一个线程安全的 `DatabaseType` 容器，并提供了向其中设置和获取 `DatabaseType` 的方法。
3. `DynamicDataSource` 继承 `AbstractRoutingDataSource` 并重写其中的方法 `determineCurrentLookupKey()`，在该方法中使用 `DatabaseContextHolder` 获取当前线程的 `DatabaseType`。
4. 在 `MyBatisConfig` 中生成两个数据源 `DataSource` 的 bean 作为第 5 步中所说的 value。
5. 在 `MyBatisConfig` 中将第 1 步中的 key 和第 4 步中的 value 组成的 key-value 对写入 `DynamicDataSource` 动态数据源的 `targetDataSources` 属性中（当然，同时也会设置两个数据源其中的一个到 `DynamicDataSource` 的 `defaultTargetDataSource` 属性中）。
6. 将 `DynamicDataSource` 作为 primary 数据源注入 `SqlSessionFactory` 的 `dataSource` 属性中。
7. 使用的时候，在 dao 层或 service 层先使用 `DatabaseContextHolder` 设置将要使用的

数据源 key (当然也可以使用 Spring AOP 去做), 然后再调用 mapper 层进行相应的操作。在 mapper 层进行操作的时候, 会先调用 `determineCurrentLookupKey()` 方法获取一个数据源 (获取数据源的方法: 先根据设置去 `targetDataSources` 中找, 若没有, 则选择 `defaultTargetDataSource`), 之后再进行数据库操作。

4.3 再学一招: MyBatis-Generator 基本用法

在本章的“再学一招”部分, 准备简单说一下 MyBatis-Generator。首先 MyBatis-Generator 的作用有如下几点:

1. 生成 model 类、mapper 接口、mapper 对应的 xml 文件。
2. 在项目的设计初期, 数据库往往会发生微小的变动, 比如添加或者修改一个字段, 但是此时我们还没怎么手动修改 mapper 接口和 xml 文件, 这个时候可以直接使用 MyBatis-Generator 重新生成以上文件, 方便数据库的修改。
3. 如果 model 类中有很多属性 (例如, 10 个以上), 自己亲手去写就有点麻烦了, 自动生成会比较快, 而且自己手写的话, 一些 jdbcType 和 javaType 的对应关系可能弄不清楚, 例如, jdbcType 是 TIMESTAMP, 那么 javaType 就是 Date, 如果设置成 java8 的 LocalDateTime 就会出错, 相应地, sessionFactory 也就无法实例化。

了解了 MyBatis-Generator 的作用之后, 看一下它的具体使用:

1. 下载 MyBatis-Generator 的配置文件、jar 包及 MySQL 的连接包 (根据不同的数据库去下不同的连接包), 下载地址: <https://github.com/mybatis/generator/releases>。
2. 准备目录结构, 如图 4-2 所示。
3. 编写配置文件, 之后运行如下命令:

```
java -jar mybatis-generator-core-1.3.2.jar -configfile generatorConfig.xml -  
overwrite
```

运行之后, 将生成的 model 类、mapper 接口及 xml 文件复制到项目中即可。

最后给出 MyBatis-Generator 的配置文件的常用选项:

```
<?xml version="1.0" encoding="UTF-8"?>  
<!DOCTYPE generatorConfiguration  
  PUBLIC "-//mybatis.org//DTD MyBatis Generator Configuration 1.0//EN"  
  "http://mybatis.org/dtd/mybatis-generator-config_1_0.dtd">
```



```

<generatorConfiguration>
<context id="mysql" targetRuntime="MyBatis3Simple" >
  <commentGenerator>
    <!-- 去除注释, 否则会在 xml 文件中生成一堆的注释信息 -->
    <property name="suppressAllComments" value="true"/>
  </commentGenerator>
  <!-- 必须要有的, 使用这个配置链接数据库 -->
  <jdbcConnection driverClass="com.mysql.jdbc.Driver"
connectionURL="jdbc:mysql://127.0.0.1:3306/microservicedb1" userId="root"
password="123456">
  </jdbcConnection>

  <!--
  java 模型创建器, 是必须要有的元素。
  targetPackage: 生成的类要放的包, 这个包名要与项目中的包名一致
  targetProject: 指定 targetPackage 包及其下生成的类存放到哪里, 注意,
Mybatis-Generator 不会自动创建该目录, 所以我们要自己手动去创建该目录
  -->
  <javaModelGenerator targetPackage="com.microservice.dbandcache.model"
targetProject="src">
  </javaModelGenerator>

  <!-- 本地缓存与分布式缓存
  用于生成 xml 映射文件
  targetPackage/targetProject: 同 javaModelGenerator
  -->
  <sqlMapGenerator targetPackage="lcw.mapping" targetProject="src">
  </sqlMapGenerator>

  <!--
  用于生成 mapper 接口
  targetPackage/targetProject: 同 javaModelGenerator
  type: 控制生成的 sql 语句写在哪儿
  1, ANNOTATEDMAPPER: 生成的 sql 语句以注解的形式写在类里;
  2, MIXEDMAPPER: 使用混合配置;
  3, XMLMAPPER: 生成的 sql 语句以注解的形式写在 xml 里; 这是最常用的格式
  -->
  <javaClientGenerator type="XMLMAPPER"
targetPackage="com.microservice.dbandcache.mapper" targetProject="src">
  </javaClientGenerator>

```

```
<!--
指定一个 table 来生成相关文件（其中也指定了 model 的类名）
tableName: 表名
domainObjectName: 生成的 domain 类的名字；
schema: 数据库的 schema，PostgreSQL 需要用到；
-->
<!--生成对应表及类名-->
<table tableName="t_user" domainObjectName="User"
enableCountByExample="false" enableUpdateByExample="false"
enableDeleteByExample="false" enableSelectByExample="false"
selectByExampleQueryId="false">
</table>
</context>
</generatorConfiguration>
```


第 5 章

微服务缓存系统

5.1 常用的缓存技术

5.1.1 本地缓存与分布式缓存

缓存一般分为本地缓存和分布式缓存两种。本地缓存指的是将数据存储在本机内存中，操作缓存数据的速度很快，但是缺点也很明显：第一，缓存数据的数量与大小受限于本机内存；第二，如果有多台应用服务器，可能所有应用服务器都要维护一份缓存，这样就占用了很多的内存。分布式缓存正好解决了这两个问题。首先，数据存储在了另外的机器上，理论上由于可以不断添加缓存机器，所以缓存的数据的数量是无限的；其次，缓存集中设置在远程的缓存服务器上，应用服务器不需要耗费空间来维护缓存。但是，分布式缓存也是有缺点的，比如由于是远程操作，所以操作缓存数据的速度相较于本地缓存慢很多。

当前用得最多的本地缓存是 GoogleGuavaCache，我们会在本章的“再学一招：使用 GuavaCache 实现本地缓存”部分对其进行演示。用得比较多的分布式缓存是 Memcached 和 Redis，下面介绍二者的应用。

5.1.2 Memcached 与 Redis

在 Redis 出现之前, Memcached 一直是分布式缓存的标配, 其使用的经典的一致性 hash 算法是缓存软件算法的标准; 基于 slab 的内存模型可以有效防止内存碎片的产生, 当然前提是设置好启动参数, 否则会浪费很多内存; 相较于 Redis2.x 版本的客户端分片技术, Memcached 的客户端分片编码会比较简单。那么, 为什么越来越多的公司开始使用 Redis 而不再使用 Memcached 呢? 首先, 相较于 Memcached 的单数据结构而言, Redis 支持 5 种数据结构: string、hash、list、set、sorted set; Redis 还提供了两种方式 (RDB 和 AOF) 来支持数据持久化, 从这一点来讲, Redis 可以被看作内存数据库; Redis 还支持事件调度、发布订阅等, 有时还可以充当一下队列, 例如在经典的 ELK 架构中, 官方就推荐使用 Redis 构建缓冲队列。下面, 介绍 Redis 的两种版本的使用方式: Redis2.x 的客户端分片和 Redis3.x 的集群。

5.2 Redis 2.x 客户端分片

5.2.1 安装 Redis

在这一部分, 笔者使用了两台操作系统是 centos7 的服务器, ip 分别是 10.211.55.10 和 10.211.55.11。Redis 使用的版本是 2.6.14。

第一步, 在开发机下载 redis-2.6.14.tar.gz, 下载地址为: <https://code.google.com/archive/p/redis/downloads?page=1>。

第二步, 将在开发机下载好的 redis-2.6.14.tar.gz 分别复制到 10.211.55.10 和 10.211.55.11。

```
scp redis-2.6.14.tar.gz root@10.211.55.10:/opt/  
scp redis-2.6.14.tar.gz root@10.211.55.11:/opt/
```

第三步, 解压安装。

```
cd /opt/  
tar -zxf redis-2.6.14.tar.gz  
cd redis-2.6.14.tar.gz  
make && make install
```

第四步, 启动服务。

```
nohup redis-server redis.conf &
```

笔者直接使用 Redis 默认的配置文​​件来启动服务，并且设置为后台启动，将相关日志写入 nohup.out 文件中。

第五步，使用 rdm (redis-desktop-manager) 软件连接 Redis。

rdm 是一款图形化的用于管理 Redis 的客户端。在其上可以方便地查询、删除 Redis 中的数据，连接信息如图 5-1 所示。

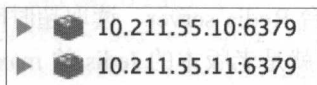


图5-1 rdm单机

5.2.2 Spring Boot 集成 ShardJedis

客户端分片通常使用 Jedis 的 ShardJedis 来实现。代码还是在 dbandcache 项目中完成，基于第 4 章的代码再添加与 Redis 有关的代码。

首先在 pom.xml 文件中引入如下依赖：

```
<!-- Jedis -->
<dependency>
    <groupId>redis.clients</groupId>
    <artifactId>jedis</artifactId>
</dependency>
<!-- 工具类 -->
<dependency>
    <groupId>org.apache.commons</groupId>
    <artifactId>commons-lang3</artifactId>
    <version>3.3.2</version>
</dependency>
<dependency>
    <groupId>com.alibaba</groupId>
    <artifactId>fastjson</artifactId>
    <version>1.1.15</version>
</dependency>
```

引入了 Jedis，在 Spring Boot 1.4.3 版本下，Jedis 的版本默认是 2.8.2，所以我们不需要指定版本。需要注意的是，太低版本的 Jedis 的 jar 包有一些 bug，所以尽量使用 2.8.0+ 的版

本。引入 commons-lang3，该包下有一系列的工具类，例如 StringUtils、NumberUtils 等，方便我们开发。引入阿里的 fastjson，该 JSON 工具类相较于 Jackson 使用起来更方便顺手。

在 src/main/resources/application.properties 中配置 Redis 信息：

```
redis.shard.servers=10.211.55.10:6379,10.211.55.11:6379
redis.shard.timeout=5000
redis.shard.maxTotal=32
```

添加 Redis 的相关信息：两台 Redis server、读取超时时间，以及能够同时建立的最大连接个数。这里的 maxTotal 其实就是老版本的 Jedis 的 maxActive，意思是能够同时建立的最大连接个数（就是最多分配多少个 ShardJedis 实例），默认为 8 个，若设置为 -1，则表示不限制个数。如果 pool 中已经分配了 maxTotal 个 ShardJedis 实例，则此时 pool 的状态就为 exhausted 的了。

配置好 Redis 之后，来看一下怎样在 Spring Boot 中集成分片版的 Jedis，com.microservice.dbandcache.config.JedisShardConfig 代码如下：

```
package com.microservice.dbandcache.config;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.core.env.Environment;
import redis.clients.jedis.JedisPoolConfig;
import redis.clients.jedis.JedisShardInfo;
import redis.clients.jedis.ShardedJedisPool;

import java.util.ArrayList;
import java.util.List;

@Configuration
public class JedisShardConfig {
    @Autowired
    private Environment env;

    @Bean
    public ShardedJedisPool shardedJedisPool() {
        String servers = env.getProperty("redis.shard.servers");
```

```

String[] serverArray = servers.split(",");//获取服务器数组
int timeout =
Integer.valueOf(env.getProperty("redis.shard.timeout"));//超时时间
JedisPoolConfig config = new JedisPoolConfig();
config.setMaxTotal(Integer.valueOf(env.getProperty(
("redis.shard.maxTotal"))));

List<JedisShardInfo> jedisList = new ArrayList<JedisShardInfo>(2);
for (String server : serverArray) {
    String[] hostAndPort = server.split(":");
    JedisShardInfo shardInfo = new JedisShardInfo(hostAndPort[0],
Integer.valueOf(hostAndPort[1]), timeout);
    jedisList.add(shardInfo);
}
return new ShardedJedisPool(config, jedisList);//构建 jedis 池
}
}

```

该类是整个客户端分片最核心的类。在这里将每一台服务器封装成一个 `JedisShardInfo`，通过这些 `JedisShardInfo` 组成的服务器列表以及 Redis 的配置信息 `JedisPoolConfig` 创建了一个 `ShardedJedisPool`。该 pool 是后续进行 Redis 操作时获取连接的地方。其中，在一个 `ShardJedisPool` 中可以同时获取多少个 `ShardJedis` 连接实例，由 `maxTotal` 配置而定。另外，笔者这里只创建了一个 `ShardJedisPool`，如果你有很多业务，而且不想让这些业务都共用几台 Redis 服务器，则可以创建多个 `ShardJedisPool`，在每个 pool 中放置不同的服务器，之后不同的业务使用不同的 `ShardJedisPool`。

然后，创建一个操作 Redis 的工具类 `com.microservice.dbandcache.util.RedisUtil`，代码如下：

```

package com.microservice.dbandcache.util;

import org.apache.commons.lang3.StringUtils;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Component;
import redis.clients.jedis.ShardedJedis;
import redis.clients.jedis.ShardedJedisPool;

@Component
public class RedisUtil {

```



```

@Autowired
private ShardedJedisPool pool;

public void set(String key, String value) {
    ShardedJedis shardedJedis = null;
    try {
        shardedJedis = pool.getResource();
        if (shardedJedis != null) {
            shardedJedis.set(key, value);
        }
    } catch (Exception e) {}
    } finally {
        if (shardedJedis != null) {
            shardedJedis.close();
        }
    }
}

public String get(String key) {
    ShardedJedis shardedJedis = null;
    try {
        shardedJedis = pool.getResource();
        if (shardedJedis != null) {
            return shardedJedis.get(key);
        }
    } catch (Exception e) {}
    } finally {
        if (shardedJedis != null) {
            shardedJedis.close();
        }
    }
    return StringUtils.EMPTY;
}
}

```

该类是操作缓存的工具类，这里笔者只写了两个操作 `String` 类型的方法：`set` 和 `get`，可以自己将其他数据结构的方法封装在该类中。看一下 `set` 和 `get` 的逻辑，首先从 `ShardedJedisPool` 中获取 `shardedJedis`，可以将此理解为操作 Redis 的一个连接，之后使用该连接进行数据库的操作。最后，无论操作成功与否，都要将 `shardedJedis` 资源关闭。

到这里为止，与业务不直接相关的代码就写完了，下面编写业务代码进行测试。首先编写一个缓存前缀指定类 `com.microservice.dbandcache.common.DbAndCacheContents`，代码如下：

```
package com.microservice.dbandcache.common;

public class DbAndCacheContents {
    public static final String USER_CACHE_PREFIX = "user:";
}
```

在该类中指定缓存前缀，指定缓存前缀最重要的作用是防止缓存 key 冲突和增强语义，一看到缓存 key 就知道该缓存是做什么用的。缓存的 key 中不同的单词之间使用“:”分隔，这是 Redis 的推荐做法。实际上，如果以“:”分隔的话，在 rdm 中我们可以看到 key 是按照“:”来分层显示的。

编写 service 层，在该层中实现缓存的调用逻辑，`com.microservice.dbandcache.service.UserService` 代码如下：

```
@Autowired
private RedisUtil redisUtil;

public User getUser(long id) {
    String userStr = redisUtil.get(DbAndCacheContents.USER_CACHE_PREFIX + id);
    if (StringUtils.isNotBlank(userStr)) {
        return JSON.parseObject(userStr, User.class);
    }
    User user = userDao.selectByPrimaryKey(id);
    if (user != null) {
        redisUtil.set(DbAndCacheContents.USER_CACHE_PREFIX + id, JSON.toJSONString(user));
    }
    return user;
}
```

这里只列出了与缓存相关的代码，该方法的具体逻辑是：首先根据用户 id 从 Redis 中获取 User 信息，如果有，直接返回给调用方；如果没有，再从数据库中查找，之后将查找到的数据添加到 Redis 中，最后返回给调用方。这也是缓存最常见的使用方法。值得注意的是，业内有一些大神推荐将缓存的调用逻辑写到 dao 层，因为缓存其实也是数据存取的

地方。

最后编写 controller, com.microservice.dbandcache.controller.DbAndCacheController 代码如下:

```
@Autowired
private UserService userService;

@ApiOperation("根据 id 获取用户信息")
@ApiImplicitParams({
    @ApiImplicitParam(paramType="query", name="id", dataType="long", required=true, value="用户的 id", defaultValue="1")
})
@ApiResponses({
    @ApiResponse(code=401, message="权限校验不通过")
})
@RequestMapping(value = "/getUserInfo", method = RequestMethod.GET)
public User getUserInfo(@RequestParam("id") long id) {
    return userService.getUser(id);
}
```

controller 依旧使用了 4.1.4 节中的按 id 查找 User 的方法。

最后, 运行代码, 使用 Swagger 进行测试。这里测试是从缓存还是从数据库中获取的数据, 可以使用断点调试的方式。

5.3 Redis 3.x 集群

客户端分片不会共享数据, 容易造成单点缓存丢失的问题; 集群会自动在多个 Redis 节点之间共享数据, 因而不会造成单点问题。所以, 使用 Redis 集群构建分布式缓存是一个不错的选择。

5.3.1 搭建 Redis 集群

建立 Redis 集群, 至少需要 3 个 master 实例, 同时, 为了使集群高可用, 需要为每个 master 实例至少分配一个 slave 实例。因此, 在这里, 为了方便, 笔者只使用 1 台操作系统是 centos7 的服务器, 在这台服务器上使用 7000~7005 6 个端口来模拟 6 个 Redis 实例, 最后将这 6 个实例组成一个三主三从的集群, 该服务器的 ip 是 10.211.55.14。这里使用的 Redis

版本是 3.2.6。

第一步，在开发机下载 redis-3.2.6.tar.gz，下载地址为：<https://redis.io/>。

第二步，将在开发机下载好的 redis-3.2.6.tar.gz 复制到 10.211.55.14。

```
scp redis-3.2.6.tar.gz root@10.211.55.14:/opt/
```

第三步，解压安装。

```
cd /opt/  
tar -zxf redis-3.2.6.tar.gz  
cd redis-3.2.6/  
make && make install
```

第四步，创建目录，复制配置文件。

```
cd /opt/  
mkdir cluster-test  
cd cluster-test/  
mkdir 7000 7001 7002 7003 7004 7005  
cp /opt/redis-3.2.6/redis.conf 7000  
cd 7000  
vi redis.conf
```

修改 7000 目录下的 redis.conf 配置。

```
bind 10.211.55.14  
port 7000  
daemonize yes  
cluster-enabled yes  
cluster-config-file nodes.conf  
cluster-node-timeout 15000  
appendonly yes
```

之后，将该配置文件分别复制到 7001~7005 中，并将配置文件中的 port 分别改为 7001~7005。该配置文件中的“bind”和“port”指定了监听的 ip 和 port；“daemonize”指定 Redis 服务是否以后台进程运行；“cluster-enabled”指定是否启用集群方式；“cluster-config-file”指定了集群配置文件，该配置文件用来存储集群的一些信息，例如集

群中各个节点的状态信息，这些信息不是让开发或运维人员编辑的。“cluster-node-timeout”指定了允许集群中的节点不可用的最大时间，例如，如果集群中的某节点超过 15s 都不可用，那么认为该节点失效了。

第五步，启动 6 个实例。

```
cd /opt/cluster-test/7000
redis-server redis.conf
```

之后，分别进入 7001~7005 5 个目录下，启动服务。

第六步，在开发机下载 redis-3.2.2.gem，下载地址为：<https://rubygems.org/gems/redis/versions/3.2.2>。

第七步，将下载好的 redis-3.2.2.gem 复制到 10.211.55.14。

```
scp redis-3.2.2.gem root@10.211.55.14:/opt/
```

第八步，安装 redis-3.2.2.gem。

```
yum install ruby rubygems -y
cd /opt/
gem install redis-3.2.2.gem
```

安装 redis-3.2.2.gem 是为了能够使用 redis-trib 命令。

第九步，创建集群。

```
cd /opt/redis-3.2.6/src/
./redis-trib.rb create --replicas 1 10.211.55.14:7000 10.211.55.14:7001
10.211.55.14:7002 10.211.55.14:7003 10.211.55.14:7004 10.211.55.14:7005
```

该步骤使用/opt/redis-3.2.6/src/下的 redis-trib.rb 命令来创建集群，其中“--replicas 1”指定一个 master 有一个 slave，之后是 6 个 Redis 实例，这就是三主三从。当该命令执行完成之后，终端显示出如下信息则表示成功。

```
[OK] All nodes agree about slots configuration.
```

第十步，使用 redis-desktop-manager 软件连接 Redis，连接信息如图 5-2 所示。

```

▶ 10.211.55.14:7000
▶ 10.211.55.14:7001
▶ 10.211.55.14:7002
▶ 10.211.55.14:7003
▶ 10.211.55.14:7004
▶ 10.211.55.14:7005

```

图5-2 rdm集群

至此，Redis 集群的搭建就成功了。

5.3.2 Spring Boot 集成 JedisCluster

通常会使用 Jedis 的 JedisCluster 来操作集群版的 Redis。代码还是在 dbandcache 项目中完成，基于第 4 章的代码再添加与 Redis 集群有关的代码。

在 pom.xml 中引入如下依赖：

```

<!-- Jedis -->
<dependency>
    <groupId>redis.clients</groupId>
    <artifactId>jedis</artifactId>
</dependency>
<!-- 工具类 -->
<dependency>
    <groupId>org.apache.commons</groupId>
    <artifactId>commons-lang3</artifactId>
    <version>3.3.2</version>
</dependency>
<dependency>
    <groupId>com.alibaba</groupId>
    <artifactId>fastjson</artifactId>
    <version>1.1.15</version>
</dependency>

```

这里引入的 3 个 jar 与 5.2.2 节中的一样。

在 src/main/resources/application.properties 文件中配置 Redis 集群信息：

```

redis.cluster.servers=10.211.55.14:7000,10.211.55.14:7001,10.211.55.14:7002,
10.211.55.14:7003,10.211.55.14:7004,10.211.55.14:7005
redis.cluster.commandTimeout=5000

```


“redis.cluster.servers”配置了集群中的 6 台机器;“redis.cluster.commandTimeout”配置了超时时间,查看方法 JedisCluster(Set<HostAndPort> nodes, int timeout)的源代码,会发现其实这个配置既是 connectionTimeout(连接超时时间),又是 soTimeout(读取超时时间)。

配置好 Redis 后,来看一下怎样在 Spring Boot 中集成 JedisCluster, com.microservice.dbandcache.config.JedisClusterConfig 代码如下:

```
package com.microservice.dbandcache.config;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.core.env.Environment;
import redis.clients.jedis.HostAndPort;
import redis.clients.jedis.JedisCluster;

import java.util.HashSet;
import java.util.Set;

@Configuration
public class JedisClusterConfig {
    @Autowired
    private Environment env;

    @Bean
    public JedisCluster jedisCluster() {
        String[] serverArray =
env.getProperty("redis.cluster.servers").split(",");
        Set<HostAndPort> nodes = new HashSet<>();
        for (String ipPort : serverArray) {
            String[] ipPortPair = ipPort.split(":");
            nodes.add(new HostAndPort(ipPortPair[0].trim(),
Integer.valueOf(ipPortPair[1].trim())));
        }
        return new JedisCluster(nodes,
Integer.valueOf(env.getProperty("redis.cluster.commandTimeout")));
    }
}
```

该类是一个核心类，在该类中，通过集群的 server 节点集合和读取超时时间创建了一个 JedisCluster 单例。之后，使用该实例直接操作 Redis 即可。

下面开始编写业务代码。com.microservice.dbandcache.service.UserService 代码如下：

```
@Autowired
private JedisCluster jedisCluster;

public User getUser(long id) {
    String userStr = jedisCluster.get(DbAndCacheContants.USER_CACHE_PREFIX
+ id);
    if (StringUtils.isNotBlank(userStr)) {
        return JSON.parseObject(userStr, User.class);
    }
    User user = userDao.selectByPrimaryKey(id);
    if (user != null) {
        jedisCluster.set(DbAndCacheContants.USER_CACHE_PREFIX + id,
JSON.toJSONString(user));
    }
    return user;
}
```

仍旧使用 5.2.2 节中的例子，只是这里不需要自己写 RedisUtil 来封装对于 Redis 的各个操作了，也不需要关心释放资源的问题了，非常简单。然后运行服务，使用 Swagger 进行测试。

5.3.3 JedisCluster 关键源码解析

在上一节中，我们说使用 JedisCluster 操作缓存，不需要自己手动释放资源，为什么呢？我们可以进入 jedisCluster.get(String key)的源码看一下，因为篇幅原因，这里只列出最关键的一个方法的源码：

```
private T runWithRetries(byte[] key, int attempts, boolean tryRandomNode,
boolean asking) {
    if(attempts <= 0) {
        throw new JedisClusterMaxRedirectionsException("Too many Cluster
redirections?");
    } else {
        Jedis connection = null;
```

```

Object var7;
try {
    if(asking) {
        connection = (Jedis)this.askConnection.get();
        connection.asking();
        asking = false;
    } else if(tryRandomNode) {
        connection = this.connectionHandler.getConnection();
    } else {
        connection = this.connectionHandler.getConnectionFromSlot
(JedisClusterCRC16.getSlot(key));
    }

    Object jre = this.execute(connection);
    return jre;
} catch (JedisNoReachableClusterNodeException var13) {
    throw var13;
} catch (JedisConnectionException var14) {
    this.releaseConnection(connection);
    connection = null;
    if(attempts <= 1) {
        this.connectionHandler.renewSlotCache();
        throw var14;
    }

    var7=this.runWithRetries(key, attempts - 1, tryRandomNode, asking);
    return var7;
} catch (JedisRedirectionException var15) {
    if(var15 instanceof JedisMovedDataException) {
        this.connectionHandler.renewSlotCache(connection);
    }

    this.releaseConnection(connection);
    connection = null;
    if(var15 instanceof JedisAskDataException) {
        asking = true;
        this.askConnection.set(this.connectionHandler.
getConnectionFromNode(var15.getTargetNode()));
    } else if(!(var15 instanceof JedisMovedDataException)) {

```

```

        throw new JedisClusterException(var15);
    }

    var7 = this.runWithRetries(key, attempts - 1, false, asking);
    } finally {
        this.releaseConnection(connection);
    }

    return var7;
}
}

```

该方法是 `JedisClusterCommand` 类的一个方法，其中入参 `attempts` 表示重试的次数（该值默认为5），通过该类我们可以看出在发生 `JedisConnectionException` 和 `JedisRedirectionException` 异常时，会进行重试，最多的重试次数为5次；并且在 `finally` 块中，执行了资源的释放，看一下这个方法：

```

private void releaseConnection(Jedis connection) {
    if(connection != null) {
        connection.close();
    }
}
}

```

就是在这里执行的关闭操作，所以在执行 `jedisCluster.get(String key)` 时，其实内部首先根据 `key` 获取了一个 `Jedis`，之后进行 `Redis` 的 `get` 操作，执行完毕之后，会将该 `Jedis` 释放掉，所以我们既不需要获取 `Jedis`，也不需要释放 `Jedis`，非常好。而且加入了重试机制，对于防止因为偶然的网络原因导致获取不到连接或重定向失败有很好的效果，而在客户端分片代码中，这些代码都要我们自己去写。

通过以上介绍，我们可以看出使用 `Redis2.x` 客户端分片方式，搭建 `Redis` 服务器非常方便，但是编写代码稍微费劲一些；而使用 `Redis3.x` 集群方式，搭建 `Redis` 集群比较麻烦，但是编写代码会非常简单！笔者推荐使用 `Redis` 集群。

5.4 再学一招：使用 GuavaCache 实现本地缓存

在本章的“再学一招”部分，笔者简单介绍一下自己觉得当下最好用的本地缓存：

GoogleGuavaCache。

1. pom.xml

```
<!-- guava cache -->
<dependency>
    <groupId>com.google.guava</groupId>
    <artifactId>guava</artifactId>
    <version>14.0.1</version>
</dependency>
```

引入 guava 的 jar 包。

2. com.microservice.dbandcache.service.UserService

```
LoadingCache<String, User> userCache = CacheBuilder.newBuilder()
    .expireAfterWrite(20, TimeUnit.MINUTES) // 缓存 20 分钟
    .maximumSize(1000) // 最多缓存 1000 个对象
    .build(new CacheLoader<String, User>() {
        public User load(String userIdKey) throws Exception {
            User user = userDao.selectByPrimaryKey(Long.valueOf(
                userIdKey.split(":")[1]));
            if (user == null) {
                user = new User();
            }
            return user;
        }
    });

public User getUser(long id) {
    User user = null;
    try {
        user = userCache.get(DbAndCacheContants.USER_CACHE_PREFIX + id);
    } catch (ExecutionException e) {
        e.printStackTrace();
    }
    return user;
}
```

在该类中，创建了一个缓存 userCache，该缓存中的缓存对象过期时间为 20 min，最多

可以缓存 1000 个对象，并且实现了抽象类 `CacheLoader` 的 `load` 方法。在该方法中，根据 `userId` 从缓存中获取 `User` 信息，最后返回 `user`。需要注意的是，该方法不可以返回 `null`。

该缓存的使用流程：首先从 `userCache` 中获取信息，如果没有，就去执行 `load` 方法，从数据库获取信息，然后自动存储到 `userCache` 中。之后再从 `userCache` 中获取 `user` 数据，传给调用方，这是不是很智能？如果使用 `Redis`，我们从数据库获取数据后，还需要手动插到 `Redis` 中。

第 6 章

Spring Boot 启动源码解析

Spring Boot 在其启动过程中给我们留出了很多的定制点，尤其是我们可以自己添加一些初始化器（`ApplicationContextInitializer` 接口的实现类）和监听器（`ApplicationListener` 接口的实现类）在服务启动的过程中执行一些逻辑，例如，可以通过监听 `ContextRefreshedEvent` 事件来实现服务启动注册，可以通过自己实现的初始化器来加载外部配置，所以了解 Spring Boot 的启动源码是非常有意义的。首先来看一下 Spring Boot 启动的正确方式：

```
public static void main(String[] args) {  
    SpringApplication sa = new SpringApplication(Application.class);  
    sa.addListeners(new ConsulRegisterListener());  
    sa.run(args);  
}
```

其中，`main` 方法中的第一行和第三行是 Spring Boot 启动主方法的最小配置。这里添加了第二行，是为了说明怎样自定义初始化器和监听器来达到我们的目的（`ConsulRegisterListener` 在下一章会用到，其监听 `ContextRefreshedEvent`，实现服务启动注册）。下面我们就来对源码做一下解析。

6.1 创建 SpringApplication 实例

首先创建 SpringApplication 类，看一下源码：

```
private List<ApplicationContextInitializer<?>> initializers;
private List<ApplicationListener<?>> listeners;
private final Set<Object> sources = new LinkedHashSet<Object>();
private boolean webEnvironment;
private Class<?> mainApplicationClass;
private static final String[] WEB_ENVIRONMENT_CLASSES =
{ "javax.servlet.Servlet",
"org.springframework.web.context.ConfigurableWebApplicationContext" };

public SpringApplication(Object... sources) {
    initialize(sources);
}

private void initialize(Object[] sources) {
    if (sources != null && sources.length > 0) {
        this.sources.addAll(Arrays.asList(sources));
    }
    this.webEnvironment = deduceWebEnvironment();
    setInitializers((Collection)
getSpringFactoriesInstances(ApplicationContextInitializer.class));
    setListeners((Collection)
getSpringFactoriesInstances(ApplicationListener.class));
    this.mainApplicationClass = deduceMainApplicationClass();
}
```

initialize 方法具体做了以下几件事：

- 将传入的“com.microservice.myseviceA.Application”放入 Set<Object> sources 集合。
- 判断是否是 Web 环境。
- 创建并初始化 ApplicationInitializer 列表。
- 创建并初始化 ApplicationListener 列表。
- 初始化主类 mainApplicationClass。

下面依次看一下每个步骤。

6.1.1 判断是否是 Web 环境

```
private boolean deduceWebEnvironment() {
    for (String className : WEB_ENVIRONMENT_CLASSES) {
        if (!ClassUtils.isPresent(className, null)) {
            return false;
        }
    }
    return true;
}
```

通过在 classpath 中查看是否存在 WEB_ENVIRONMENT_CLASSES 这个数组中所包含的所有类（实际上就是 2 个类：javax.servlet.Servlet 和 org.springframework.web.context.ConfigurableWebApplicationContext），如果两个类都存在那么当前程序即是一个 Web 应用程序，反之则不然。我们这里是 true。

6.1.2 创建并初始化 ApplicationInitializer 列表

```
public void setInitializers(Collection<? extends
ApplicationContextInitializer<?>> initializers) {
    this.initializers = new ArrayList<ApplicationContextInitializer<?>>();
    this.initializers.addAll(initializers);
}
```

创建一个 ArrayList<ApplicationContextInitializer<?>> 实例 initializers，之后将所有传入的初始化器都添加到 initializers 实例中。

这些 initializer 是怎么获取的呢？看一下源码：

```
private <T> Collection<? extends T> getSpringFactoriesInstances(Class<T> type) {
    return getSpringFactoriesInstances(type, new Class<?>[] {});
}

private <T> Collection<? extends T> getSpringFactoriesInstances(Class<T> type,
    Class<?>[] parameterTypes, Object... args) {
    ClassLoader classLoader = Thread.currentThread().getContextClassLoader();
    // Use names and ensure unique to protect against duplicates
    Set<String> names = new LinkedHashSet<String>();
    SpringFactoriesLoader.loadFactoryNames(type, classLoader);
    List<T> instances = createSpringFactoriesInstances(type, parameterTypes,
```

```

classLoader, args, names);
AnnotationAwareOrderComparator.sort(instances);
return instances;
}

```

其中最重要的就两句，获取要加载的 `initializer` 的名字（并且使用 `set` 集合去重）；之后将所有的 `initializer` 实例化，并返回。

实例化的代码没什么好说的，就是简单反射创建对象，代码如下：

```

private <T> List<T> createSpringFactoriesInstances(Class<T> type,
Class<?>[] parameterTypes, ClassLoader classLoader, Object[] args,
Set<String> names) {
List<T> instances = new ArrayList<T>(names.size());
for (String name : names) {
try {
Class<?> instanceClass = ClassUtils.forName(name, classLoader);
Assert.isAssignable(type, instanceClass);
Constructor<?> constructor = instanceClass
.getDeclaredConstructor(parameterTypes);
T instance = (T) BeanUtils.instantiateClass(constructor, args);
instances.add(instance);
}
catch (Throwable ex) {
throw new IllegalArgumentException(
"Cannot instantiate " + type + " : " + name, ex);
}
}
return instances;
}

```

关键是怎么获取这些 `initializer` 的名字呢？来看一下 `SpringFactoriesLoader.loadFactoryNames(type, classLoader)` 的源码：

```

public static final String FACTORIES_RESOURCE_LOCATION =
"META-INF/spring.factories";
public static List<String> loadFactoryNames(Class<?> factoryClass, ClassLoader
classLoader) {
String factoryClassName = factoryClass.getName();
try {

```



```

Enumeration<URL> urls = (classLoader != null ?
classLoader.getResources(FACTORIES_RESOURCE_LOCATION) :
ClassLoader.getSystemResources(FACTORIES_RESOURCE_LOCATION));
List<String> result = new ArrayList<String>();
while (urls.hasMoreElements()) {
URL url = urls.nextElement();
Properties properties = PropertiesLoaderUtils.loadProperties(new UrlResource
(url));
String factoryClassNames = properties.getProperty(factoryClassName);
result.addAll(Arrays.asList(StringUtils.commaDelimitedListToStringArray
(factoryClassNames)));
}
return result;
}
catch (IOException ex) {
throw new IllegalArgumentException("Unable to load [" + factoryClass.getName()
+
"] factories from location [" + FACTORIES_RESOURCE_LOCATION + "]", ex);
}
}

```

其实就是从 Maven 仓库中的 `spring-boot-1.4.3.RELEASE.jar` 以及 `spring-boot-autoconfigure-1.4.3.RELEASE.jar` 的 `META-INF/spring.factories` 中获取 key 为 `org.springframework.context.ApplicationContextInitializer` 的所有 `initializers` 全类名。看一下 `spring-boot-1.4.3.RELEASE.jar` 中的 `spring.factories` 中的 `initializers`:

```

# Application Context Initializers
org.springframework.context.ApplicationContextInitializer=\
org.springframework.boot.context.ConfigurationWarningsApplicationContextInit
ializer,\
org.springframework.boot.context.ContextIdApplicationContextInitializer,\
org.springframework.boot.context.config.DelegatingApplicationContextInitiali
zer,\
org.springframework.boot.context.web.ServerPortInfoApplicationContextInitial
izer

```

再看一下 `spring-boot-autoconfigure-1.4.3.RELEASE.jar` 中的 `spring.factories` 中的 `initializers`:

```

# Initializers
org.springframework.context.ApplicationContextInitializer=\

```

```
org.springframework.boot.autoconfigure.SharedMetadataReaderFactoryContextInitializer,\norg.springframework.boot.autoconfigure.logging.AutoConfigurationReportLoggingInitializer
```

所以，初始化后的 `List<ApplicationContextInitializer<?>> initializers` 包含如上 6 个 `initializer` 实例。

6.1.3 创建并初始化 `ApplicationListener` 列表

```
public void setListeners(Collection<? extends ApplicationListener<?>> listeners) {\n    this.listeners = new ArrayList<ApplicationListener<?>>();\n    this.listeners.addAll(listeners);\n}
```

初始化 `List<ApplicationListener<?>> listeners` 实例的过程与初始化 `initializer` 实例的过程一样。来看一下 `spring-boot-1.4.3.RELEASE.jar` 中的 `spring.factories` 中的 `listeners`:

```
# Application Listeners\norg.springframework.context.ApplicationListener=\norg.springframework.boot.ClearCachesApplicationListener,\norg.springframework.boot.builder.ParentContextCloserApplicationListener,\norg.springframework.boot.context.FileEncodingApplicationListener,\norg.springframework.boot.context.config.AnsiOutputApplicationListener,\norg.springframework.boot.context.config.ConfigFileApplicationListener,\norg.springframework.boot.context.config.DelegatingApplicationListener,\norg.springframework.boot.liquibase.LiquibaseServiceLocatorApplicationListener,\norg.springframework.boot.logging.ClasspathLoggingApplicationListener,\norg.springframework.boot.logging.LoggingApplicationListener
```

再看一下 `spring-boot-autoconfigure-1.4.3.RELEASE.jar` 中的 `spring.factories` 中的 `listeners`:

```
# Application Listeners\norg.springframework.context.ApplicationListener=\norg.springframework.boot.autoconfigure.BackgroundPreinitializer
```

所以，初始化后的 `List<ApplicationListener<?>> listeners` 包含如上 10 个 `listener` 实例。

6.1.4 初始化主类 mainApplicationClass

代码如下：

```
private Class<?> deduceMainApplicationClass() {
    try {
        StackTraceElement[] stackTrace = new RuntimeException().getStackTrace();
        for (StackTraceElement stackTraceElement : stackTrace) {
            if ("main".equals(stackTraceElement.getMethodName())) {
                return Class.forName(stackTraceElement.getClassName());
            }
        }
    } catch (ClassNotFoundException ex) {
    }
    return null;
}
```

该方法也很简单，首先获取方法调用栈，之后循环遍历该调用栈，最后一个 `stackTraceElement` 的方法名就是 `main`（因为方法是从 `main` 方法发起的）。

初始化完主类之后，`Class<?> mainApplicationClass` 的值为 `com.microservice.myServiceA.Application`。

6.2 添加自定义监听器

看一下添加自定义监听器的正确方法：

```
sa.addListeners(new ConsulRegisterListener());
```

看一下源代码：

```
public void addListeners(ApplicationListener<?>... listeners) {
    this.listeners.addAll(Arrays.asList(listeners));
}
```

就是向之前的 `listener` 列表中添加一个自己的监听器。添加一个 `initializer` 的方法与添加 `listener` 相似。

6.3 启动核心 run 方法

最后来看 `sa.run(args)`方法做了什么？源码如下：

```
public ConfigurableApplicationContext run(String... args) {
    Stopwatch stopWatch = new Stopwatch();
    stopWatch.start();
    ConfigurableApplicationContext context = null;
    FailureAnalyzers analyzers = null;
    configureHeadlessProperty();
    SpringApplicationRunListeners listeners = getRunListeners(args);
    listeners.started();
    try {
        ApplicationArguments applicationArguments = new DefaultApplicationArguments(
            args);
        ConfigurableEnvironment environment = prepareEnvironment(listeners,
            applicationArguments);
        Banner printedBanner = printBanner(environment);
        context = createApplicationContext();
        analyzers = new FailureAnalyzers(context);
        prepareContext(context, environment, listeners, applicationArguments,
            printedBanner);
        refreshContext(context);
        afterRefresh(context, applicationArguments);
        listeners.finished(context, null);
        stopWatch.stop();
        if (this.logStartupInfo) {
            new StartupInfoLogger(this.mainApplicationClass)
                .logStarted(getApplicationLog(), stopWatch);
        }
        return context;
    }
    catch (Throwable ex) {
        handleRunFailure(context, listeners, analyzers, ex);
        throw new IllegalStateException(ex);
    }
}
```

其实就是做了如下几件事：

- 创建计时器 Stopwatch。
- 配置 awt 系统属性。
- 获取 SpringApplicationRunListeners。
- 启动 SpringApplicationRunListener。
- 创建 ApplicationArguments。
- 创建并初始化 ConfigurableEnvironment。
- 打印 Banner。
- 创建 ConfigurableApplicationContext。
- 准备 ConfigurableApplicationContext。
- 刷新 ConfigurableApplicationContext。
- 容器刷新后动作。
- SpringApplicationRunListeners 发布 finish 事件。
- 计时器停止计时。

6.3.1 创建启动停止计时器

首先把计时器的部分介绍一下，看一下源代码：

```
private final String id;
private boolean running;
private String currentTaskName;
private long startTimeMillis;
private long totalTimeMillis;

public Stopwatch() {
    this("");
}

public Stopwatch(String id) {
    this.id = id;
}

public void start() throws IllegalStateException {
    start("");
}

public void start(String taskName) throws IllegalStateException {
```



```

if (this.running) {
    throw new IllegalStateException("Can't start Stopwatch: it's already running");
}
this.running = true;
this.currentTaskName = taskName;
this.startTimeMillis = System.currentTimeMillis();
}

public void stop() throws IllegalStateException {
    if (!this.running) {
        throw new IllegalStateException("Can't stop Stopwatch: it's not running");
    }
    long lastTime = System.currentTimeMillis() - this.startTimeMillis;
    this.totalTimeMillis += lastTime;
    this.lastTaskInfo = new TaskInfo(this.currentTaskName, lastTime);
    if (this.keepTaskList) {
        this.taskList.add(lastTaskInfo);
    }
    ++this.taskCount;
    this.running = false;
    this.currentTaskName = null;
}

```

这里只列出了关键的代码，可以看到 `start` 方法只是做了记录当前时间的工作；在 `stop` 中计算了总的启动时间。

6.3.2 配置 awt 系统属性

设置 awt 系统属性，与我们的服务关系不大，只列出源码：

```

private static final String SYSTEM_PROPERTY_JAVA_AWT_HEADLESS = "java.awt.
headless";
private boolean headless = true;

private void configureHeadlessProperty() {
    System.setProperty(SYSTEM_PROPERTY_JAVA_AWT_HEADLESS, System.getProperty(
SYSTEM_PROPERTY_JAVA_AWT_HEADLESS, Boolean.toString(this.headless)));
}

```

6.3.3 获取 SpringApplicationRunListeners

来看一下源代码：

```
private SpringApplicationRunListeners getRunListeners(String[] args) {
    Class<?>[] types = new Class<?>[] { SpringApplication.class, String[].class };
    return new SpringApplicationRunListeners(logger, getSpringFactoriesInstances(
        SpringApplicationRunListener.class, types, this, args));
}
```

与之前初始化 `initializer` 和 `listener` 一样，从 `spring-boot-1.4.3.RELEASE.jar` 中的 `spring.factories` 中读取 `SpringApplicationRunListeners` 全类名，之后创建该对象。来看一下 `spring.factories` 中有哪些 `SpringApplicationRunListeners`：

```
# Run Listeners
org.springframework.boot.SpringApplicationRunListener=\
org.springframework.boot.context.event.EventPublishingRunListener
```

实际上就是创建了一个仅含 `EventPublishingRunListener` 实例的列表。之后将该列表赋值到 `SpringApplicationRunListeners` 类中的 `listeners` 实例中，代码如下：

```
private final Log log;
private final List<SpringApplicationRunListener> listeners;

SpringApplicationRunListeners(Log log,
    Collection<? extends SpringApplicationRunListener> listeners) {
    this.log = log;
    this.listeners = new ArrayList<SpringApplicationRunListener>(listeners);
}
```

`EventPublishingRunListener` 类是一个非常重要的类，用来发布事件，触发监听器。我们来看一下该对象的创建源码：

```
private final SpringApplication application;
private final String[] args;
private final ApplicationEventMulticaster initialMulticaster;

public EventPublishingRunListener(SpringApplication application, String[] args) {
    this.application = application;
    this.args = args;
```

```

this.initialMulticaster = new SimpleApplicationEventMulticaster();
for (ApplicationListener<?> listener : application.getListeners()) {
    this.initialMulticaster.addApplicationListener(listener);
}
}

```

这里将之前所有的 listener 都添加到了 initialMulticaster 实例中,这样,当 initialMulticaster 发布事件时,监听器监听到该事件,就会做出相应的动作。

6.3.4 启动 SpringApplicationRunListener

源码如下:

```

public void started() {
    for (SpringApplicationRunListener listener : this.listeners) {
        listener.started();
    }
}

```

此时 EventPublishingRunListener 就会发布 ApplicationStartedEvent 事件:

```

@Override
public void started() {
    this.initialMulticaster
        .multicastEvent(new ApplicationStartedEvent(this.application, this.args));
}

```

所有监听了该事件的 listener 就会执行相应的方法。这里不列出来了。

6.3.5 创建 ApplicationArguments

代码如下:

```

ApplicationArguments applicationArguments = new DefaultApplicationArguments
(args);

```

其中传入的 args 参数就是 main 方法中传入的参数。这里传入的是--spring.output.ansi.enabled=always, 该参数指定了日志输出使用多彩输出。

看一下源码:

```
private final Source source;
private final String[] args;

public DefaultApplicationArguments(String[] args) {
    Assert.notNull(args, "Args must not be null");
    this.source = new Source(args);
    this.args = args;
}
```

其中 Source 内部类是 SimpleCommandLinePropertySource 的子类, SimpleCommandLinePropertySource 是 CommandLinePropertySource 的子类, CommandLinePropertySource 是 EnumerablePropertySource 的子类, EnumerablePropertySource 是 PropertySource 的子类。如果把源代码列出来,会很占篇幅,这里只给出最后 DefaultApplicationArguments 的初始化结果(形象化的表示,不考虑是否符合 Java 语法),两个属性: args 数组和 Source source。其中 args 数组如下:

```
String[] args = {"--spring.output.ansi.enabled=always"};
```

Source source 如下:

```
String name = "commandLineArgs";
T source = commandLineArgs;
```

再来看一下 CommandLineArgs:

```
private final Map<String, List<String>> optionArgs = new HashMap<String,
List<String>>();
private final List<String> nonOptionArgs = new ArrayList<String>();
```

CommandLineArgs 包含以上两个参数,其中 optionArgs 在代码执行之后,包含一个 key-value 对,其中 key 为“spring.output.ansi.enabled”,value 是一个只包含一个元素“always”的列表。nonOptionArgs 没有变化。

6.3.6 创建并初始化 ConfigurableEnvironment

代码如下:

```
ConfigurableEnvironment environment = prepareEnvironment(listeners,
applicationArguments);
```

看一下源码:

```
private ConfigurableEnvironment prepareEnvironment(
    SpringApplicationRunListeners listeners,
    ApplicationArguments applicationArguments) {
    // Create and configure the environment
    ConfigurableEnvironment environment = getOrCreateEnvironment();
    configureEnvironment(environment, applicationArguments.getSourceArgs());
    listeners.environmentPrepared(environment);
    if (isWebEnvironment(environment) && !this.webEnvironment) {
        environment = convertToStandardEnvironment(environment);
    }
    return environment;
}
```

总的来说，一共做了三件事：创建一个 `ConfigurableEnvironment` 实例；配置 `ConfigurableEnvironment` 实例；发布 `ApplicationEnvironmentPreparedEvent` 事件，通知所有监听了该事件的监听器做相应的操作。

1. 创建 `ConfigurableEnvironment` 实例

```
private ConfigurableEnvironment getOrCreateEnvironment() {
    if (this.environment != null) {
        return this.environment;
    }
    if (this.webEnvironment) {
        return new StandardServletEnvironment();
    }
    return new StandardEnvironment();
}
```

创建 `ConfigurableEnvironment` 实例时，由于是 Web 环境，因此 `this.webEnvironment` 为 `true`，所以创建了一个 `StandardServletEnvironment`。在调用 `StandardEnvironment` 的构造器的时候，会先调用其父类 `AbstractEnvironment` 的构造器，代码如下：

```
private final MutablePropertySources propertySources = new
MutablePropertySources(this.logger);

public AbstractEnvironment() {
    customizePropertySources(this.propertySources);
}
```



```

if (this.logger.isDebugEnabled()) {
    this.logger.debug(String.format(
        "Initialized %s with PropertySources %s", getClass().getSimpleName(),
        this.propertySources));
}
}
}

```

其中，`MutablePropertySources` 是 `PropertySources` 的子类。其含有一个很重要的属性，如下：

```

private final List<PropertySource<?>> propertySourceList = new
CopyOnWriteArrayList<PropertySource<?>>();

```

`propertySourceList` 属性会存放所有的属性源 `PropertySource`。接下来就做这个事儿。在 `AbstractEnvironment` 的无参构造器中调用 `StandardServletEnvironment` 的 `customizePropertySources` 方法，源码如下：

```

protected void customizePropertySources(MutablePropertySources
propertySources) {
    propertySources.addLast(new StubPropertySource(SERVLET_CONFIG_PROPERTY_
SOURCE_NAME));
    propertySources.addLast(new StubPropertySource(SERVLET_CONTEXT_PROPERTY_
SOURCE_NAME));
    if (JndiLocatorDelegate.isDefaultJndiEnvironmentAvailable()) {
        propertySources.addLast(new JndiPropertySource(JNDI_PROPERTY_SOURCE_NAME));
    }
    super.customizePropertySources(propertySources);
}

```

这里为 `propertySourceList` 添加了两个 `PropertySource`：一个是 name 为“`servletConfigInitParams`”的属性源；另一个是 name 为“`servletContextInitParams`”的属性源。这两个属性源都暂且没有属性值。之后调用了 `StandardServletEnvironment` 的父类的 `customizePropertySources` 方法，源码如下：

```

/** System environment property source name: {@value} */
public static final String SYSTEM_ENVIRONMENT_PROPERTY_SOURCE_NAME =
"systemEnvironment";
/** JVM system properties property source name: {@value} */
public static final String SYSTEM_PROPERTIES_PROPERTY_SOURCE_NAME =
"systemProperties";

```

```
protected void customizePropertySources(MutablePropertySources propertySources) {
    propertySources.addLast(new MapPropertySource(SYSTEM_PROPERTIES_PROPERTY_
SOURCE_NAME, getSystemProperties()));
    propertySources.addLast(new SystemEnvironmentPropertySource(SYSTEM_ENVIRONMENT_
PROPERTY_SOURCE_NAME, getSystemEnvironment()));
}
```

这里为 `propertySourceList` 又添加两个 `PropertySource`，一个是 `MapPropertySource`，其 `name` 为“`systemProperties`”，值为一个 `Map<String, Object>` 的数据结构，主要用于存放 JVM 的一些系统属性，例如，“`java.runtime.name`” - “`Java(TM) SE Runtime Environment`” 这样的 `key-value` 对。值得注意的是，其中在启动的 JVM 参数中设置了“`-Dserver.port=8088`”，这个值就被存在了该 `Map` 中，`server.port=8088`。此时，里边暂时存了 55 个 `key-value` 对。

另一个属性源是 `SystemEnvironmentPropertySource`，其是一个 `name` 为“`system-Environment`”，值为 `Map<String,String>` 的数据结构，主要用于存放一些系统环境属性，通常我们配置的环境变量，例如在 `~/bash_profile` 等文件中设置的都会被读取到，比如 `M2_HOME` 等。里边暂时存放了 21 个 `key-value` 对。

此时，一个 `ConfigurableEnvironment` 实例就创建成功了！看一下创建完成后该实例包含的一些东西：

- 一个不包含任何元素的 `LinkedHashSet`，`name` 为“`activeProfiles`”。
- 一个包含一个元素“`default`”的 `LinkedHashSet`，`name` 为“`defaultProfiles`”。
- 一个 `MutablePropertySources` 实例 `propertySources`，其内部的属性 `propertySourceList` 包含 4 个属性源。
- 一个 `PropertySourcesPropertyResolver` 属性，主要用于解析属性源。其内部包含一个包含上述 `MutablePropertySources` 实例的属性，以及用于解析属性源的其他属性，例如 `placeholderPrefix=${}`、`placeholderSuffix=}`、`valueSeparator=` 等。

2. 配置 `ConfigurableEnvironment` 实例

```
protected void configureEnvironment(ConfigurableEnvironment environment,
String[] args) {
    configurePropertySources(environment, args);
    configureProfiles(environment, args);
}
```

对于 `ConfigurableEnvironment` 的配置，其实就是配置其两个属性：`propertySources` 和

activeProfiles。

首先配置 `propertySources`:

```
private boolean addCommandLineProperties = true;

protected void configurePropertySources(ConfigurableEnvironment environment,
    String[] args) {
    MutablePropertySources sources = environment.getPropertySources();
    if (this.defaultProperties != null && !this.defaultProperties.isEmpty()) {
        sources.addLast(
            new MapPropertySource("defaultProperties", this.defaultProperties));
    }
    if (this.addCommandLineProperties && args.length > 0) {
        String name = CommandLinePropertySource.COMMAND_LINE_PROPERTY_SOURCE_NAME;
        if (sources.contains(name)) {
            PropertySource<?> source = sources.get(name);
            CompositePropertySource composite = new CompositePropertySource(name);
            composite.addPropertySource(new SimpleCommandLinePropertySource(
                name + "-" + args.hashCode(), args));
            composite.addPropertySource(source);
            sources.replace(name, composite);
        }
        else {
            sources.addFirst(new SimpleCommandLinePropertySource(args));
        }
    }
}
```

该方法传入的参数 `args` 是包含一个参数“`--spring.output.ansi.enabled=always`”的数组，所以整个流程就是创建了一个 `SimpleCommandLinePropertySource`（其实该属性源就是读取 `main` 函数的传入参数并做相应的封装），并将该属性源放置在 `ConfigurableEnvironment` 的 `propertySources` 属性的首部，也就是说该属性源的优先级最高。`propertySources` 属性中的属性源的存放位置从前到后优先级依次降低。此时 `propertySources` 属性中包含 5 个属性源。

再配置 `activeProfiles`:

```
protected void configureProfiles(ConfigurableEnvironment environment, String[]
args) {
    environment.getActiveProfiles(); // ensure they are initialized
    // But these ones should go first (last wins in a property key clash)
```

```

Set<String> profiles = new LinkedHashSet<String>(this.additionalProfiles);
profiles.addAll(Arrays.asList(environment.getActiveProfiles()));
environment.setActiveProfiles(profiles.toArray(new
String[profiles.size()]));
}

```

该配置文件其实就是读取“spring.profiles.active”所指定的配置文件，该配置值通常用于指定在不同的环境下使用不同的配置文件，但是在配置外放之后几乎不会再使用该配置了。这里我们没有配置它，所以 activeProfiles 属性的 size 值依然为零。可以简单地看一下 environment.getActiveProfiles() 的源码：

```

public static final String ACTIVE_PROFILES_PROPERTY_NAME = "spring.profiles.active";

@Override
public String[] getActiveProfiles() {
    return StringUtils.toStringArray(doGetActiveProfiles());
}

protected Set<String> doGetActiveProfiles() {
    synchronized (this.activeProfiles) {
        if (this.activeProfiles.isEmpty()) {
            String profiles = getProperty(ACTIVE_PROFILES_PROPERTY_NAME);
            if (StringUtils.hasText(profiles)) {
                setActiveProfiles(StringUtils.commaDelimitedListToStringArray(
                    StringUtils.trimAllWhitespace(profiles)));
            }
        }
        return this.activeProfiles;
    }
}

```

3. 发布 ApplicationEnvironmentPreparedEvent 事件

```
listeners.environmentPrepared(environment);
```

看一下上边这行代码的源码：

```

private final List<SpringApplicationRunListener> listeners;

public void environmentPrepared(ConfigurableEnvironment environment) {
    for (SpringApplicationRunListener listener : this.listeners) {

```



```
listener.environmentPrepared(environment);
}
}
```

在 listeners 中只有一个元素，就是之前的 EventPublishingRunListener，此时该监听器发布了 ApplicationEnvironmentPreparedEvent 事件。看一下源码：

```
@Override
public void environmentPrepared(ConfigurableEnvironment environment) {
    this.initialMulticaster.multicastEvent(new ApplicationEnvironmentPreparedEvent(
        this.application, this.args, environment));
}
```

所有监听了该事件的监听器执行相应的逻辑。这里不列出了。这些监听器监听之后，最主要做一件事，就是读取配置文件 application.properties，并把这些信息存储在一个 name 为 “applicationConfigurationProperties” 的属性源中，并将该属性源置于整个属性源列表的最后。如果将来只是从外放的配置文件中读取配置，最好将该配置源删掉，然后自己读取外部配置文件构建数据源，之后再将该数据源放到数据源列表中。

6.3.7 打印 Banner

```
Banner printedBanner = printBanner(environment);
```

该行主要用于打印出 Spring Boot 的图标和版本号，可以自己定制，这里不做解析。

6.3.8 创建 ConfigurableApplicationContext

```
context = createApplicationContext();
```

该行代码主要用于创建一个 ApplicationContext，源码如下：

```
/**
 * The class name of application context that will be used by default for non-web
 * environments.
 */
public static final String DEFAULT_CONTEXT_CLASS =
    "org.springframework.context."
    + "annotation.AnnotationConfigApplicationContext";
/**
```



```

* The class name of application context that will be used by default for web
* environments.
*/
public static final String DEFAULT_WEB_CONTEXT_CLASS = "org.springframework."
    + "boot.context.embedded.AnnotationConfigEmbeddedWebApplicationContext";

protected ConfigurableApplicationContext createApplicationContext() {
    Class<?> contextClass = this.applicationContextClass;
    if (contextClass == null) {
        try {
            contextClass = Class.forName(this.webEnvironment
                ? DEFAULT_WEB_CONTEXT_CLASS : DEFAULT_CONTEXT_CLASS);
        }
        catch (ClassNotFoundException ex) {
            throw new IllegalStateException(
                "Unable create a default ApplicationContext, "
                + "please specify an ApplicationContextClass",
                ex);
        }
    }
    return (ConfigurableApplicationContext)
        BeanUtils.instantiate(contextClass);
}

```

这里主要是为 `contextClass` 赋值, 使用了 Spring Boot 自己的一个 `AnnotationConfigEmbeddedWebApplicationContext`。之后使用反射实例化该 `ApplicationContext`, 看一下 `instantiate` 的源码:

```

public static <T> T instantiate(Class<T> clazz) throws BeanInstantiationException {
    Assert.notNull(clazz, "Class must not be null");
    if (clazz.isInterface()) {
        throw new BeanInstantiationException(clazz, "Specified class is an interface");
    }
    try {
        return clazz.newInstance();
    }
    catch (InstantiationException ex) {
        throw new BeanInstantiationException(clazz, "Is it an abstract class?", ex);
    }
    catch (IllegalAccessException ex) {

```

```

        throw new BeanInstantiationException(clazz, "Is the constructor accessible?", ex);
    }
}

```

6.3.9 准备 ConfigurableApplicationContext

```
prepareContext(context, environment, listeners, applicationArguments, printedBanner);
```

看一下源码：

```

private void prepareContext(ConfigurableApplicationContext context,
    ConfigurableEnvironment environment, SpringApplicationRunListeners listeners,
    ApplicationArguments applicationArguments, Banner printedBanner) {
    context.setEnvironment(environment);
    postProcessApplicationContext(context);
    applyInitializers(context);
    listeners.contextPrepared(context);
    if (this.logStartupInfo) {
        logStartupInfo(context.getParent() == null);
        logStartupProfileInfo(context);
    }

    // Add boot specific singleton beans
    context.getBeanFactory().registerSingleton("springApplicationArguments",
        applicationArguments);
    if (printedBanner != null) {
        context.getBeanFactory().registerSingleton("springBootBanner", printedBanner);
    }

    // Load the sources
    Set<Object> sources = getSources();
    Assert.notEmpty(sources, "Sources must not be empty");
    load(context, sources.toArray(new Object[sources.size()]));
    listeners.contextLoaded(context);
}

```

考虑到篇幅的问题，以下只分析重要的源码。以上比较重要的代码有：执行初始化器；加载配置。

1. 执行初始化器

```
applyInitializers(context);
```

看一下源代码：

```
protected void applyInitializers(ConfigurableApplicationContext context) {
    for (ApplicationContextInitializer initializer : getInitializers()) {
        Class<?> requiredType = GenericTypeResolver.resolveTypeArgument(
            initializer.getClass(), ApplicationContextInitializer.class);
        Assert.isInstanceOf(requiredType, context, "Unable to call initializer.");
        initializer.initialize(context);
    }
}
```

遍历所有的初始化器，执行每一个初始化器的 `initialize` 方法。这里可以通过实现 `ApplicationContextInitializer<ConfigurableApplicationContext>` 接口来创建自己的初始化器。例如，可以在初始化器中读取外部配置文件。

2. 加载配置

```
load(context, sources.toArray(new Object[sources.size()]));
```

其中，`sources` 是一个只包含主类的 `set` 集合。之后将该主类注册到 `AnnotationConfigEmbeddedWebApplicationContext` 的 `beanFactory` 属性中。`beanFactory` 属性是 `AnnotationConfigEmbeddedWebApplicationContext` 的父类 `GenericApplicationContext` 中的一个属性，源码如下：

```
private final DefaultListableBeanFactory beanFactory;
```

而在 `DefaultListableBeanFactory` 中含有如下属性：

```
private final Map<String, BeanDefinition> beanDefinitionMap = new
ConcurrentHashMap<String, BeanDefinition>(256);
```

实际上最后会将主类注册到 `beanDefinitionMap` 中。`key` 为 “application”，`value` 是一个主类的单例 `Bean`。`load` 方法调用链较长，其中的一个方法比较重要，如下：

```
private int load(Class<?> source) {
    if (isGroovyPresent()) {
        // Any GroovyLoaders added in beans{} DSL can contribute beans here
        if (GroovyBeanDefinitionSource.class.isAssignableFrom(source)) {
```

```

GroovyBeanDefinitionSource loader = BeanUtils.instantiateClass(source,
    GroovyBeanDefinitionSource.class);
load(loader);
}
}
if (isComponent(source)) {
    this.annotatedReader.register(source);
    return 1;
}
return 0;
}

private boolean isComponent(Class<?> type) {
    // This has to be a bit of a guess. The only way to be sure that this type is
    // eligible is to make a bean definition out of it and try to instantiate it.
    if (AnnotationUtils.findAnnotation(type, Component.class) != null) {
        return true;
    }
    // Nested anonymous classes are not eligible for registration, nor are groovy
    // closures
    if (type.getName().matches(".*\\$_.*closure.*") || type.isAnonymousClass()
        || type.getConstructors() == null || type.getConstructors().length == 0) {
        return false;
    }
    return true;
}
}

```

所以主类需要加上@Component注解。

6.3.10 刷新 ConfigurableApplicationContext

```
refreshContext(context);
```

调用链也比较长，比较重要的是 AbstractApplicationContext 的 refresh()方法：

```

public void refresh() throws BeansException, IllegalStateException {
    Object var1 = this.startupShutdownMonitor;
    synchronized(this.startupShutdownMonitor) {
        this.prepareRefresh();
        ConfigurableListableBeanFactory beanFactory =
this.obtainFreshBeanFactory();

```



```

        this.prepareBeanFactory(beanFactory);

        try {
            this.postProcessBeanFactory(beanFactory);
            this.invokeBeanFactoryPostProcessors(beanFactory);
            this.registerBeanPostProcessors(beanFactory);
            this.initMessageSource();
            this.initApplicationEventMulticaster();
            this.onRefresh();
            this.registerListeners();
            this.finishBeanFactoryInitialization(beanFactory);
            this.finishRefresh();
        } catch (BeansException var9) {
            if (this.logger.isWarnEnabled()) {
                this.logger.warn("Exception encountered during context
initialization - cancelling refresh attempt: " + var9);
            }

            this.destroyBeans();
            this.cancelRefresh(var9);
            throw var9;
        } finally {
            this.resetCommonCaches();
        }
    }
}

```

这个方法是 Spring 最重要的一个方法，其中比较重要的有 `this.onRefresh()` 和 `this.finishRefresh()` 方法。

1. onRefresh

`onRefresh` 的调用链比较长，比较重要的是：

```

@Override
protected void onRefresh() {
    super.onRefresh();

    try {
        createEmbeddedServletContainer();
    }
}

```



```

catch (Throwable ex) {
    throw new ApplicationContextException("Unable to start embedded container",
        ex);
}
}

```

该方法创建了一个内嵌的 Servlet 容器,用于执行 Web 应用,默认使用 TomcatEmbeddedServletContainer。

2. finishRefresh

```

@Override
protected void finishRefresh() {
    super.finishRefresh();
    EmbeddedServletContainer localContainer = startEmbeddedServletContainer();
    if (localContainer != null) {
        publishEvent(
            new EmbeddedServletContainerInitializedEvent(this, localContainer));
    }
}

```

首先调用了父类 AbstractApplicationContext 的 finishRefresh()方法,在该方法中发布了 ContextRefreshedEvent 事件,所有监听了该事件的监听器开始执行逻辑;然后启动 TomcatEmbeddedServletContainer,最后发布 EmbeddedServletContainerInitializedEvent,所有监听了该事件的监听器执行相应的逻辑。父类的 finishRefresh()方法源码如下:

```

protected void finishRefresh() {
    this.initLifecycleProcessor();
    this.getLifecycleProcessor().onRefresh();
    this.publishEvent((ApplicationEvent) (new ContextRefreshedEvent(this)));
    LiveBeansView.registerApplicationContext(this);
}

```

6.3.11 容器刷新后动作

```
afterRefresh(context, applicationArguments);
```

该方法几乎没做什么事,不做解析了。

6.3.12 SpringApplicationRunListeners 发布 finish 事件

```
listeners.finished(context, null);
```

这里发布 `ApplicationReadyEvent` 或 `ApplicationFailedEvent` 事件，所有监听了该事件的监听器去执行相应的逻辑。

6.3.13 计时器停止计时

```
stopWatch.stop();
```

这个方法之前分析过，这里就不分析了。

6.4 再学一招：常用的获取属性的 4 种方法

假设在 `application.properties` 中配置了以下两个属性：

```
userinfo.name=nana
userinfo.age=18
```

可以使用 4 种方法来获取这两个属性值。分别来看一下。

1. @Value

```
@Value("${userinfo.name}")
private String username;
```

2. Environment

```
@Autowired
private Environment env;

public String getUsername(){
    return env.getProperty("userinfo.name");
}
```

3. 整体代换

构造一个新类，专门用于读取配置。

```
import org.springframework.boot.context.properties.ConfigurationProperties;
import org.springframework.stereotype.Component;
```

```
@Component
@ConfigurationProperties(prefix = "userinfo")
public class UserProperties {
    private String name;
    private int age;

    public String getName() {
        return name;
    }

    public int getAge() {
        return age;
    }

    public void setName(String name) {
        this.name = name;
    }

    public void setAge(int age) {
        this.age = age;
    }
}
```

之后，将该类作为一个 Bean 注入使用类中。如下：

```
@Autowired
private UserProperties userProperties;

public String getUserInfo() {
    return userProperties.getName()+"-->"+userProperties.getAge();
}
```

4. 动态获取

这种方式主要是使用第三方的工具包提供的获取属性方式，例如使用 Archaius 获取配置信息。具体见“配置管理”部分。

7.2 搭建 Consul 集群

在本书环境中，推荐 Consul 集群使用 3 台或 5 台 serverAgent，这里这里为了方便，只搭建了一台 serverAgent，ip 是 10.211.55.12，此外，还搭建了 2 台 clientAgent，ip 分别是 10.211.55.13 和 10.211.55.14。

第 7 章

7.2.1 安装 Consul

第 2 篇

服务框架篇

服务框架篇

可以直接在上面的 3 台 centos 上执行 wget https://releases.hashicorp.com/consul/0.7.2/consul_0.7.2_linux_amd64.zip -O consul.zip 下载，但是，这种下载方式需要看网络速度。

因此，将下载好的 consul_0.7.2_linux_amd64.zip 复制到一台 centos 服务器上。

第 7 章 微服务注册与发现

第 8 章 微服务配置管理

第 9 章 微服务进程间通信

第 10 章 微服务降级容错

第 7 章

微服务注册与发现

7.1 初识 Consul

Consul 是一个分布式、高可用、支持多数据中心的软件，主要用于服务注册、服务发现、key/value 存储及健康检查等。

在实际使用中，为了实现高可用，需要搭建 Consul 集群。在整个 Consul 集群中，所有的 Consul 节点分为两种角色：serverAgent 和 clientAgent。每个 Consul 集群中至少需要有一个 serverAgent，官方推荐 3 或 5 台。serverAgent 会存储一些集群的状态信息，并响应集群中其他 agent 的相关操作。通常在我们的应用服务集群中的每一个节点上都会有一个 clientAgent，通过该 agent 将当前节点上的服务向 Consul 进行注册，该 agent 也负责该服务节点及其上的服务的健康检查。Consul 配合 Spring Boot 的 Actuator 可以非常简单地实现健康检查（只需要引入 spring-boot-starter-actuator），所以可以很方便地找出健康的节点供调用方使用。Consul 的 key/value 存储功能也很重要，该功能可用于动态配置、leader 选举等。Consul 还提供了非常漂亮的 UI，用来进行 key/value、节点、服务及 session（可充当分布式锁）等的相关操作。

7.2 搭建 Consul 集群

在生产环境中，推荐 Consul 集群使用 3 台或 5 台 serverAgent，笔者这里为了方便，只搭建了一台 serverAgent，ip 是 10.211.55.12。此外，还搭建了两台 clientAgent，ip 分别是 10.211.55.13 和 10.211.55.14。

7.2.1 安装 Consul

第一步，在开发机下载 Consul。

下载地址为：<https://www.consul.io/downloads.html>。笔者下载了 Linux 64 位的 Consul。还可以直接在上边的三台 centos 上执行 `wget https://releases.hashicorp.com/consul/0.7.2/consul_0.7.2_linux_amd64.zip -O consul.zip` 下载。但是，这种下载方式需要看网络速度。

第二步，将下载好的 `consul_0.7.2_linux_amd64.zip` 复制到三台 centos 机器上。

```
scp consul_0.7.2_linux_amd64.zip root@10.211.55.12:/opt/
scp consul_0.7.2_linux_amd64.zip root@10.211.55.13:/opt/
scp consul_0.7.2_linux_amd64.zip root@10.211.55.14:/opt/
```

第三步，分别进入三台 centos，解压缩。

```
cd /opt/
unzip consul_0.7.2_linux_amd64.zip
```

之后将解压出来的二进制文件 Consul 移动到 `/usr/bin/` 下。

```
mv consul /usr/bin/consul
```

Consul 安装成功后，启动各个 Consul 节点。

7.2.2 启动 Consul 集群

第一步，在 10.211.55.12 处启动 serverAgent。

```
nohup consul agent -server -data-dir=/tmp/consul -node=server-12
-bind=10.211.55.12 -bootstrap-expect 1 -client 0.0.0.0 -ui -dc=zjgdc &
```

说明：笔者将 Consul 以后台进程来运行，并且将相关日志写在 `nohup.out` 文件中，整条命令比较长，下面逐一介绍每个命令和选项。

- `consul agent`: 表示该命令会启动一个 `consulAgent`。
- `-server`: 表示该 `agent` 是一个 `serverAgent`, 不添加这个选项的话, 表示是一个 `clientAgent`。
- `-data-dir`: 表示相关数据存储的目录位置, 在 `serverAgent` 上该目录下会存储一些集群的状态信息, 而在 `clientAgent` 上主要存储在其上注册的服务信息以及这些服务的健康检查信息。
- `-node`: 指定该 `agent` 节点的名称, 该名称在集群中必须是唯一的 (默认采用机器的 `host`)。
- `-bind`: 指定该 `agent` 的 `ip`。
- `-bootstrap-expect 1`: 该命令通知 Consul 我们现在准备加入的 `server` 节点个数, 该参数是为了延迟日志复制的启动, 直到指定数量的 `server` 节点成功加入后才启动。
- `-client 0.0.0.0 -ui`: 启动 Consul-UI, 如果不添加 “`-client 0.0.0.0`” 选项, 则 UI 只能在当前机器上 (即 `10.211.55.12`) 进行访问。
- `-dc`: 指定该 `agent` 加入哪一个数据中心, 默认是 `dc1`。

第二步, 在 `10.211.55.13` 和 `10.211.55.14` 处分别启动 `clientAgent` 并加入到集群中。

在 `10.211.55.13` 上执行:

```
nohup consul agent -data-dir=/tmp/consul -node=client-13 -bind=10.211.55.13  
-join=10.211.55.12 -dc=zjgdc &
```

在 `10.211.55.14` 上执行:

```
nohup consul agent -data-dir=/tmp/consul -node=client-14 -bind=10.211.55.14  
-join=10.211.55.12 -dc=zjgdc &
```

下面逐一介绍每个命令和选项。

- `consul agent`: 同上;
- `-data-dir`: 同上;
- `-node`: 同上;
- `-bind`: 同上;
- `-join`: 将节点加入集群;
- `-dc`: 同上。

这里启动的时候，我们没有指定-server，默认以 clientAgent 来启动。

第三步，验证集群是否搭建成功。在三台机器的任意一台上执行：

```
consul members
```

该命令用于列出集群中的所有节点，执行之后，若结果如图 7-1 所示，表示搭建成功。

```
[root@centos7-2 opt]# consul members
Node      Address          Status  Type   Build  Protocol  DC
client-13  10.211.55.13:8301 alive   client 0.7.2   2         zjgdc
client-14  10.211.55.14:8301 alive   client 0.7.2   2         zjgdc
server-12  10.211.55.12:8301 alive   server 0.7.2   2         zjgdc
```

图7-1 Consul节点信息

7.2.3 启动 Consul-UI

在上一节中，在启动 serverAgent 时启动了 Consul-UI。下面，打开 Consul-UI 查看相关信息。

在浏览器中输入：<http://10.211.55.12:8500>，并单击其中的 Nodes 标签，结果如图 7-2 所示。

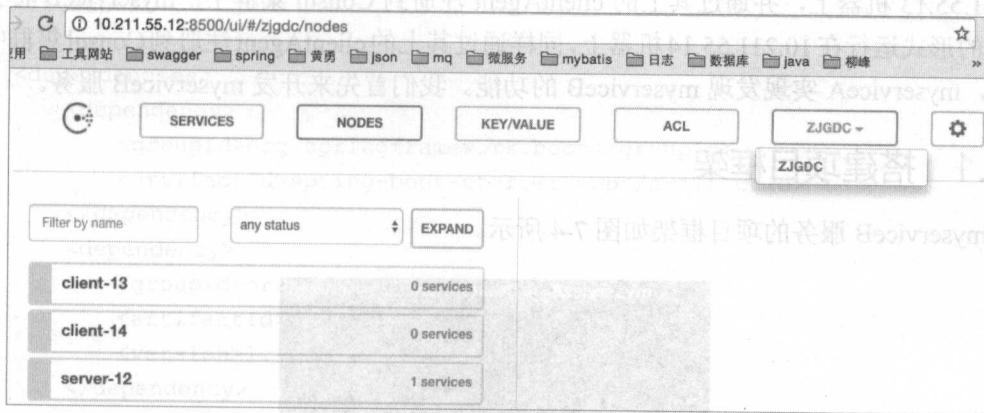


图7-2 Consul-UI信息

Consul-UI 中有 5 个导航按钮，其中 SERVICES 导航下包含所有注册到 Consul 上的服务；NODES 导航下包含所有 Consul 集群中的节点机器；KEY/VALUE 导航下包含所有服务的配置信息；ACL 导航下包含权限控制信息；最后一个导航按钮用于选择数据中心。

到此，Consul 集群就搭建完成了。下面，我们来编写代码实现服务注册、发现与健康

检查等功能。

7.3 使用 Consul 实现服务注册与服务发现

本节将搭建的集群架构如图 7-3 所示。

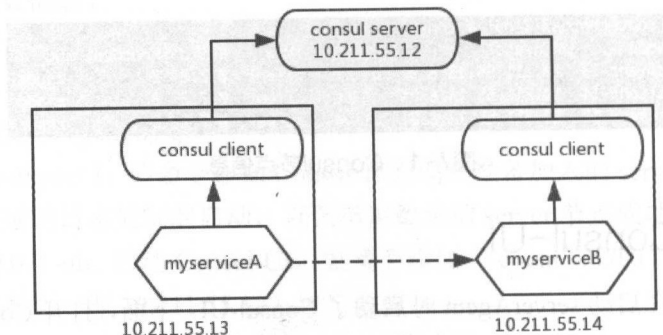


图7-3 Consul集群架构图

将创建两个服务：myserviceA 和 myserviceB，myserviceA 将会以 jar 包的形式运行在 10.211.55.13 机器上，并通过其上的 clientAgent 注册到 Consul 集群中；myserviceB 将会以 jar 包的形式运行在 10.211.55.14 机器上，同样通过其上的 clientAgent 注册到 Consul 集群中。最后，myserviceA 实现发现 myserviceB 的功能。我们首先来开发 myserviceB 服务。

7.3.1 搭建项目框架

myserviceB 服务的项目框架如图 7-4 所示。

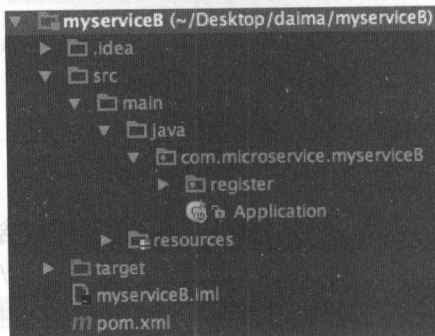


图7-4 myserviceB项目结构

其中，pom.xml 文件内容如下：

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/maven-v4_0_0.xsd">
  <modelVersion>4.0.0</modelVersion>

  <groupId>com.microservice</groupId>
  <artifactId>myserviceB</artifactId>
  <version>1.0-SNAPSHOT</version>

  <properties>
    <java.version>1.8</java.version>
  </properties>
  <parent>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-parent</artifactId>
    <version>1.4.3.RELEASE</version>
  </parent>

  <dependencies>
    <dependency>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-starter-web</artifactId>
    </dependency>
    <dependency>
      <groupId>org.projectlombok</groupId>
      <artifactId>lombok</artifactId>
      <version>1.16.8</version>
    </dependency>
    <!-- consul-client -->
    <dependency>
      <groupId>com.orbitz.consul</groupId>
      <artifactId>consul-client</artifactId>
      <version>0.13.8</version>
    </dependency>
    <!-- consul 需要的包 -->
    <dependency>
```



```
<groupId>org.glassfish.jersey.core</groupId>
<artifactId>jersey-client</artifactId>
<version>2.22.2</version>
</dependency>
<dependency>
<groupId>org.springframework.boot</groupId>
<artifactId>spring-boot-starter-actuator</artifactId>
</dependency>
</dependencies>

<build>
<plugins>
<plugin>
<groupId>org.springframework.boot</groupId>
<artifactId>spring-boot-maven-plugin</artifactId>
</plugin>
</plugins>
</build>
</project>
```

在该 pom.xml 中，引入了 spring-boot-starter-web、Lombok、consul-client、jersey-client 及 spring-boot-starter-actuator 5 个依赖。Consul 官网给出了两种 Java 客户端：consul-client 和 consul-api，可以选择其一，这里，笔者使用 consul-client。为了使用 consul-client，引入了 consul-client 和 jersey-client 两个依赖；为了实现健康检查，引入了 spring-boot-starter-actuator 依赖，该依赖是 Spring Boot 可用于生产级别的监控工具，包含进行健康检查以及收集 JVM 的相关统计数据的 metrics。在微服务计数监控系统中，可以通过对这些统计数据进行分析告警。

项目框架搭建完成之后，可以编写程序来实现服务注册功能。

7.3.2 配置服务注册信息

在实现服务注册之前，首先来配置服务注册信息。在 application.properties 中配置信息，内容如下：

```
service.name=myServiceB
service.port=8080
service.tag=dev
health.url=http://localhost:${service.port}/health
```

```
health.interval=10
```

`service.name` 指定了服务名, 我们通常使用服务的 `artifactId` 来命名; `service.port` 指定服务将运行在哪个端口; `service.tag` 指定服务的标签, 该选项主要用于区分环境, 例如是开发环境还是线上环境, 通常使用 `dev` 表示开发环境, `prod` 表示线上环境; `health.url` 是 Consul 进行健康检查的 `url`, Consul 会以一个指定的频率访问这个 `url`, 如果显示的是 UP 的值, 则服务健康, 否则, 服务不健康; 这个指定的频率就是 `health.interval` 的值, 这里是 10 ms。这 5 个参数是服务向 Consul 注册服务的最基本的值。

配置好注册信息后, 为了将这些零散的配置信息整顿起来方便后续使用, 笔者编写了一个 Bean 来读取并封装这些参数。该 Bean 内容如下:

```
package com.microservice.myserviceB.register;

import org.springframework.beans.factory.annotation.Value;
import org.springframework.stereotype.Component;

import lombok.Getter;
import lombok.Setter;

@Component
@Getter
@Setter
public class ConsulProperties {
    @Value("${service.name}")
    private String servicename;
    @Value("${service.port:8080}")
    private int    servicePort;
    @Value("${service.tag:dev}")
    private String serviceTag;
    @Value("${health.url}")
    private String healthUrl;
    @Value("${health.interval:10}")
    private int    healthInterval;
}
```

这样, 使用者将该 Bean 注入其他类中直接使用即可。其中, 如果 `service.port` 的值没有配置, 则默认取 8080; 如果 `service.tag` 的值没有配置, 则默认取 `dev`。

配置好后, 接下来实现本章最核心的第一个任务: 服务启动注册!

7.3.3 实现服务启动注册

什么是服务注册？引用第 1 章中的一段话，“服务注册简单形象地来讲就是将服务的 ip 和 port 注册到注册中心，这里可以简单地将注册中心理解为一个 Map，其中的 key 是服务的唯一标识（可以是 serviceID，也可以是 serviceName），而 value 是一个包含 ipAndPort 结构体的集合，例如是一个 List 集合，该 List 集合中存放了指定 service 所在的所有服务器。”

下面不仅要实现服务注册功能，还要实现在服务启动时自动注册到 Consul。首先定义一个 Consul 单例 Bean。代码如下：

```
package com.microservice.myserviceB.register;

import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;

import com.orbitz.consul.Consul;

@Configuration
public class ConsulConfig {
    @Bean
    public Consul consul() {
        return Consul.newClient();
    }
}
```

通常显式创建 Bean 的类时会起名为 XxxConfig。为了实现服务启动时自动注册到 Consul，我们需要清楚地知道 Spring Boot 的启动流程。下面先看一下代码，然后再做分析。代码如下：

```
package com.microservice.myserviceB.register;

import java.net.MalformedURLException;
import java.net.URI;

import org.springframework.context.ApplicationListener;
import org.springframework.context.event.ContextRefreshedEvent;

import com.orbitz.consul.Consul;
```

```

public class ConsulRegisterListener implements
ApplicationListener<ContextRefreshedEvent>{
    @Override
    public void onApplicationEvent (ContextRefreshedEvent contextRefreshedEvent) {
        Consul consul =
contextRefreshedEvent.getApplicationContext().getBean(Consul.class);
        ConsulProperties consulProperties = contextRefreshedEvent.
getApplicationContext().getBean(ConsulProperties.class);
        try {
            consul.agentClient().register(consulProperties.getServicePort(),
                                         URI.create(consulProperties.getHealth-
Url()).toURL(),
                                         consulProperties.getHealthInterval(),
                                         consulProperties.getServiceName(),
                                         consulProperties.getServiceName(),
                                         consulProperties.getServiceTag());
        } catch (MalformedURLException e) {
            e.printStackTrace();
        }
    }
}

```

ConsulRegisterListener 类实现了 `org.springframework.context.ApplicationListener<T>`，并且将泛型 T 指定为 `ContextRefreshedEvent`，这样该类就会监听容器刷新事件。当 Spring Boot 发布该事件时，ConsulRegisterListener 将会执行 `onApplicationEvent` 方法，在该方法内，调用 Consul 的 `AgentClient` 实现服务注册。在注册过程中，指定了服务端口、健康检查 url、健康检查的时间间隔、服务名、服务 ID 及服务标签。其中，服务 ID 是在注册中心上识别服务的唯一标识，不能重复，这里将服务名作为了服务 ID。

值得注意的是，我们没有把 ConsulRegisterListener 这个类注册为 Bean，这里提供了一种在非 Bean 类调用 Bean 的方法：先想办法获取 `ApplicationContext`，然后再使用其 `getBean` 方法获取 Bean。

最后，将 ConsulRegisterListener 注册到 SpringBoot 的启动监听器列表中去，实现服务启动注册功能。该步骤在服务启动类中完成，代码如下：

```

package com.microservice.myserviceA;

import com.microservice.myserviceA.register.ConsulRegisterListener;

```



```
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import springfox.documentation.swagger2.annotations.EnableSwagger2;

@SpringBootApplication
@EnableSwagger2
public class Application {
    public static void main(String[] args) {
        SpringApplication sa = new SpringApplication(Application.class);
        sa.addListeners(new ConsulRegisterListener());
        sa.run(args);
    }
}
```

之前说过，要实现服务启动时自动注册到 Consul，我们需要清楚地知道 SpringBoot 的启动流程，这是很重要的。Spring Boot 的启动流程可以参考第 6 章 Spring Boot 启动源码解析部分内容。

至此，服务启动注册功能就完成了。我们再来开发 myserviceA 服务，在 myserviceA 中实现服务发现功能。

7.3.4 实现服务发现

myserviceA 项目结构如图 7-5 所示。

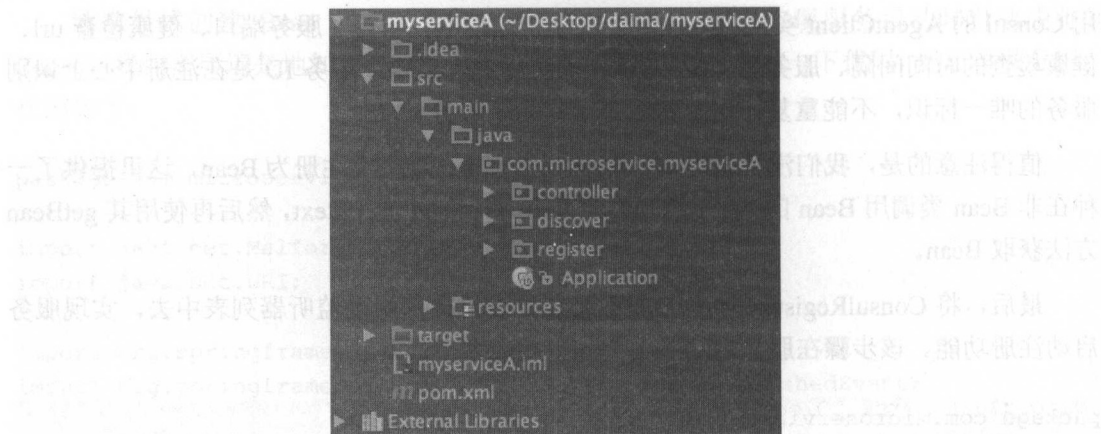


图7-5 myserviceA 项目结构

myserviceA 要实现服务发现功能，通常也需要注册到 Consul，注册代码与 myserviceB 几乎相同，这里不再赘述。本节只列出实现服务发现功能的代码。什么是服务发现？在第 1 章中曾解释过，服务发现简单来讲就是根据服务的唯一标识，从注册中心获取指定服务的服务器列表。

首先，定义一个服务器模型，代码如下：

```
package com.microservice.myserviceA.discover;
```

```
import lombok.AllArgsConstructor;
```

```
import lombok.Getter;
```

```
import lombok.Setter;
```

```
@Getter
```

```
@Setter
```

```
@AllArgsConstructor
```

```
public class Server {
```

```
    private String host;
```

```
    private int port;
```

```
}
```

Server 类只包含 host 和 port 两个信息，为了后续使用方便，这里使用 Lombok 注解 @AllArgsConstructor 来生成全参构造器。

下面，实现第二个核心任务：服务发现。代码如下：

```
package com.microservice.myserviceA.discover;
```

```
import com.orbitz.consul.Consul;
```

```
import com.orbitz.consul.model.health.ServiceHealth;
```

```
import org.springframework.beans.factory.annotation.Autowired;
```

```
import org.springframework.stereotype.Component;
```

```
import java.util.ArrayList;
```

```
import java.util.List;
```

```
@Component
```

```
public class ServiceDiscovery {
```

```
    @Autowired
```

```
    private Consul consul;
```

```

public List<Server> getServerList(String serviceName) {
    List<Server> upServerList = new ArrayList<>();
    List<ServiceHealth> availableServers = consul.healthClient().
getHealthyServiceInstances(serviceName).getResponse();
    availableServers.forEach(x -> upServerList.add(new
Server(x.getNode().getAddress(), x.getService().getPort())));
    return upServerList;
}
}

```

ServiceDiscovery 类的代码非常简单，通过 Consul 的 HealthClient 根据 serviceName 获取注册在 Consul 上的健康的服务列表。

7.4 服务部署测试

7.4.1 编写测试类

为了测试服务发现功能，在 myserviceA 服务中编写一个 controller 类，代码如下：

```

package com.microservice.myserviceA.controller;

import com.microservice.myserviceA.discover.Server;
import com.microservice.myserviceA.discover.ServiceDiscovery;
import io.swagger.annotations.*;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestMethod;
import org.springframework.web.bind.annotation.RequestParam;
import org.springframework.web.bind.annotation.RestController;

import java.util.List;

@Api("测试 api")
@RestController
@RequestMapping("/test")
public class TestController {
    @Autowired
    private ServiceDiscovery serviceDiscovery;
}

```

```

@ApiOperation("根据 serviceName 获取服务列表")
@ApiImplicitParams({ @ApiImplicitParam(paramType = "query", name =
"serviceName", dataType = "string", required = true, value = "服务名称") })
@RequestMapping(value = "/getServiceList", method = RequestMethod.GET)
public List<Server> getServiceList(@RequestParam("serviceName") String
serviceName) {
    return serviceDiscovery.getServerList(serviceName);
}
}

```

由于使用了 Swagger, 因此在 myserviceA 的 pom.xml 中需要引入 Swagger 的两个依赖: springfox-swagger2 和 springfox-swagger-ui, 并且在启动类上添加@EnableSwagger2。Swagger 的用法参考第3章相关内容。

一切就绪之后, 开始部署服务!

7.4.2 服务打包部署

第一步, 分别将 myserviceA 和 myserviceB 打成 jar 包。

分别在 myserviceA 和 myserviceB 的根目录, 也就是根 pom.xml 存在的目录下执行 mvn clean install。之后, 在二者的 target 下看到 myserviceA-1.0-SNAPSHOT.jar 和 myserviceB-1.0-SNAPSHOT.jar。

第二步, 将 myserviceB-1.0-SNAPSHOT.jar 上传到 10.211.55.13 机器并启动, 将 myserviceA-1.0-SNAPSHOT.jar 上传到 10.211.55.14 机器并启动。

```

scp myserviceB/target/myserviceB-1.0-SNAPSHOT.jar root@10.211.55.13:/opt/
scp myserviceB/target/myserviceA-1.0-SNAPSHOT.jar root@10.211.55.14:/opt/

```

之后在 10.211.55.13 机器上执行以下命令来启动 myserviceB 服务:

```
nohup java -jar myserviceB-1.0-SNAPSHOT.jar &
```

之后在 10.211.55.14 机器上执行以下命令来启动 myserviceA 服务:

```
nohup java -jar myserviceA-1.0-SNAPSHOT.jar &
```

值得注意的是, 我们的服务需要的 JDK 版本至少是 1.8, 如果虚拟机上的 JDK 版本低于 1.8, 那么需要先把老的 JDK 卸载掉, 再安装 JDK1.8, 之后再启动服务。这里, 笔者简单地介绍一下, 怎样卸载老的 JDK 并安装 JDK1.8。以 10.211.55.13 为例, 在其上执行: rpm

-qa | grep java, 然后将显示出来的东西全部卸载掉。

```
rpm -e --nodeps python-javapackages-3.4.1-11.el7.noarch
rpm -e --nodeps tzdata-java-2016j-1.el7.noarch
rpm -e --nodeps javapackages-tools-3.4.1-11.el7.noarch
rpm -e --nodeps java-1.7.0-openjdk-headless-1.7.0.121-2.6.8.0.el7_3.x86_64
rpm -e --nodeps java-1.7.0-openjdk-1.7.0.121-2.6.8.0.el7_3.x86_64
```

这样, 就将原本的 JDK1.7 卸载掉了。之后, 安装 JDK1.8。

在本机下载好 jdk-8u102-linux-x64.tar.gz 后, 上传到 10.211.55.13, 之后解压, 配置环境变量即可。

```
tar -xvf jdk-8u102-linux-x64.tar.gz
vi /etc/profile
```

在/etc/profile 中添加如下内容:

```
JAVA_HOME=/opt/jdk1.8.0_102
export JAVA_HOME
export PATH=$PATH:$JAVA_HOME/bin
```

之后执行 source /etc/profile 使得/etc/profile 文件在不重启服务器的情况下生效, 然后用 java -version 查看是否安装成功。

第三步, 在本地的 Consul-UI 上查看服务注册情况, 如图 7-6 所示。



图7-6 service注册情况

由图 7-6 可见, myserviceA 和 myserviceB 注册成功!

7.4.3 运行测试

在 10.211.55.14 上执行以下命令测试服务发现:

```
curl -X GET "http://localhost:8080/test/getServiceList?serviceName=myserviceB"
```

出现 [{"host": "10.211.55.13", "port": 8080}] 表明, 服务发现功能正常运行!!!
当然也可以通过 Swagger 进行访问。

在实际开发中, 服务打包部署是不需要手动去操作的。只需要使用 GitLab、Jenkins、Maven、简单的 Shell 脚本以及 webhook 就可以实现服务自动化编译、打包、复制和部署。在第 15 章中, 我们将会搭建这套服务部署系统。

7.5 使用 Consul 与 Actuator 实现健康检查

7.5.1 健康检查机制

在 myserviceA 和 myserviceB 中, 我们引入了 spring-boot-starter-actuator 依赖包, 该依赖提供了基本的健康检查功能, 引入该依赖后, 通过访问 “http://ip:port/health” 这个 url, 可以查看部署在 ip:port 上的服务的健康状况。通常全部显示为 “UP” 就是健康的。

在服务注册的时候, Consul 本身进行健康检查时所访问的 url 就是上边配置的这个 url。这样就简单方便地实现了健康检查功能。

7.5.2 健康检查查错思路

在图 7-6 中, 如果发现 myserviceA 或者 myserviceB 不是绿色 passing 的, 而是黄色 failing 的, 如图 7-7 所示, 那么 myserviceA 服务就不健康了, 此时就需要看一下健康检查的信息了。查看的基本思路是: 首先查看是不是服务所在节点 node 跪了, 如果节点 node 没有跪, 那么再去检查服务为什么跪了。



图7-7 myserviceA服务不健康

查看节点是否跪了有两种方式：第一种，直接通过 Consul UI，单击 myserviceA，如图 7-8 所示，其中，Serf Health Status 是绿的，证明节点没跪，如果是黄色的，证明节点跪了。

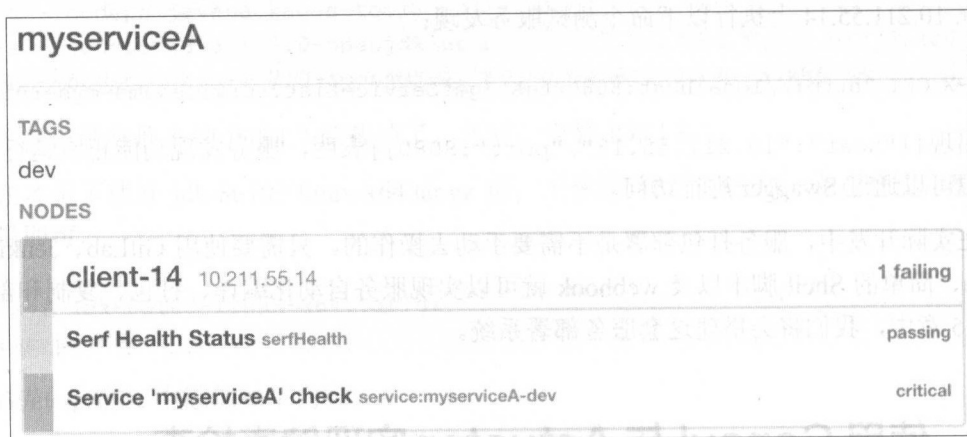


图7-8 Consul健康检查

第二种，在 10.211.55.14 上执行 `consul members` 看 client agent 是否是 alive 状态。

如果通过上述检查，发现节点没有跪，那么是服务跪了，通过执行 `curl -X GET "http://ip:port/health"`，看一下哪些组件的状态是 DOWN 而不是 UP 的。之后，去做相应的处理即可。如果访问链接之后发现干脆没有响应，那么看一下是不是没有引入 Actuator 包或者服务根本没有启动。

7.6 再学一招：Consul 健康检查分类及原理

前边说过，consul agent 的一个主要作用就是实现系统级别和应用级别的健康检查，也就是说健康检查包含两部分：首先，会对 agent 所在节点的状态进行检查；其次，agent 会对其上的应用服务的状态进行检查。Consul 的健康检查目前为止有 5 种实现方式：Script + Interval、HTTP + Interval、TCP + Interval、Time to Live (TTL) 以及 Docker + Interval。其中，最常用最简单的方式就是第二种 HTTP + Interval，这也是笔者所使用的方式。

HTTP + Interval 的实现方式其实就是对一个指定的 url 每隔一定的时间执行一次 GET 请求。GET 请求返回的 `httpCode` 是 2xx，表示健康检查通过，服务健康 (passing)；如果是 429，表示请求数太多，服务警告 (warning)。除了这两类 `httpCode` 之外，其他的都表示服

务不健康 (failure)。默认情况下, 健康检查的 http 请求的超时时间与请求间隔时间是相等的, 最大值是 10 s, 当然我们也可以在健康检查定义中自己指定一个时间。看一下 HTTP + Interval 方式的健康检查定义格式:

```
{
  "check": {
    "id": "api",
    "name": "HTTP API on port 8080",
    "http": "http://localhost:8080/health",
    "interval": "10s",
    "timeout": "1s"
  }
}
```

该定义文件在服务启动之后, 会在从其下启动 consul agent 命令的 -data-dir 目录下生成。

第 8 章

微服务配置管理

8.1 初识 Archaius

Archaius 是 Netflix 开源技术栈中的一员，其提供了一系列与配置管理相关的 API。

8.1.1 为什么要使用 Archaius

在上一章中我们实现了服务注册与服务发现，发现了服务之后，就可以使用一定的服务路由技术，并通过一定的通信技术，进行服务之间的调用了。不过在做这些之前，笔者准备先解决一个问题，就是“配置问题”。就目前情况而言，假设现在需要修改 myserviceA 一个配置，我们需要怎么做呢？首先，在本机的 myserviceA 服务的 application.properties 文件中修改配置，之后需要先编译打包代码，再将 jar 包传到相关的服务器上，最后再启动服务。即使我们使用了持续集成与持续部署系统，这些操作也一定是要执行的，只不过由机器来执行而已，仍旧需要花费时间。这是一个很麻烦的事情！那么，有没有一种方式，当我们修改了配置文件后，不需要执行上边这个流程，就可以使修改后的配置信息直接生效呢？答案是肯定的。本章将会通过 Netflix 的 Archaius 和 Consul-KV 来实现这个功能。

8.1.2 Archaius 原理

Archaius 名字来源于变色龙的一个种类，所以也有人将其翻译成“变色龙”，取名为“变

色龙”的原因是因为变色龙可以根据环境或情况的不同来动态地改变自己的颜色，而设计 Archaius 这个框架的目的就是想要不重启服务而获取修改后的配置信息。Archaius 具体有如下关键的特征：

- 提供了一个 polling 框架来实现“热配置”（即在不重启服务器的情况下获取修改后的配置信息）。
- 配置管理操作全部都是线程安全的。
- 可以自己提供配置源，即从哪里获取配置信息，如数据库、Consul-KV 等。
- Archaius 还支持多配置源，并且通过内部的一个层级结构来管理这些配置源。

8.2 使用 Consul-KV 实现配置集中管理

在上一章讲过，Consul 有一个可用于管理配置文件的功能，叫作 key/value store，简称为 Consul-KV。Consul 的 UI 界面提供了对 Consul-KV 的增删改查功能。这里通过 Consul-UI 创建了一个 key 为“service/config/dev”、value 为“animal.name=pig”的 key-value 对。如图 8-1 所示。



图8-1 Consul-KV配置

接下来，编写代码来读取这个配置并且使用 Archaius 来实现动态配置。

8.3 使用 Archaius 实现动态获取配置

为了不让知识点耦合在一起，笔者在之后的每一章都会尽量新建一个微服务来完成当前章节所讲的知识点（除非一定要用到之前所讲的技术），当看完整本书后，读者可以自己将这些知识点总结在一起，那么一个完整的微服务框架就形成了。

8.3.1 搭建项目框架

这里新建一个微服务 config，项目结构如图 8-2 所示。

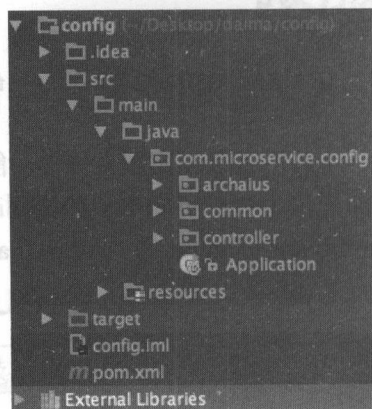


图8-2 config项目结构

其中，pom.xml 文件内容如下：

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/maven-v4_0_0.xsd">
  <modelVersion>4.0.0</modelVersion>

  <groupId>com.microservice</groupId>
  <artifactId>config</artifactId>
  <version>1.0-SNAPSHOT</version>

  <properties>
    <java.version>1.8</java.version>
```



```
</properties>
<parent>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-parent</artifactId>
  <version>1.4.3.RELEASE</version>
</parent>

<dependencies>
  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
  </dependency>
  <!-- archaius -->
  <dependency>
    <groupId>com.netflix.archaius</groupId>
    <artifactId>archaius-core</artifactId>
    <version>0.7.5</version>
  </dependency>
  <!-- 动态配置, Archaius 底层 -->
  <dependency>
    <groupId>commons-configuration</groupId>
    <artifactId>commons-configuration</artifactId>
    <version>1.8</version>
  </dependency>
  <!-- consul-client -->
  <dependency>
    <groupId>com.orbitz.consul</groupId>
    <artifactId>consul-client</artifactId>
    <version>0.13.8</version>
  </dependency>
  <!-- consul 需要的包 -->
  <dependency>
    <groupId>org.glassfish.jersey.core</groupId>
    <artifactId>jersey-client</artifactId>
    <version>2.22.2</version>
  </dependency>
  <!-- 工具类 -->
  <dependency>
    <groupId>org.apache.commons</groupId>
    <artifactId>commons-lang3</artifactId>
```

```

        <version>3.3.2</version>
    </dependency>
</dependencies>

<build>
    <plugins>
        <plugin>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-maven-plugin</artifactId>
        </plugin>
    </plugins>
</build>
</project>

```

在该 pom.xml 中, 引入了 spring-boot-starter-web、consul-client、jersey-client、archaius-core、commons-configuration、commons-lang3 以及 spring-boot-starter-actuator 7 个依赖。引入了 consul-client 和 jersey-client, 用于操作 Consul, 从 Consul-KV 中读取配置信息; 引入了 archaius-core 和 commons-configuration, 用于实现配置信息的动态获取, 其中 Archaius 是伟大的 Netflix 开源技术栈中的一员, 专门用于动态读取配置信息, 实现“热配置”功能, 其底层使用的技术是 Apache 的 commons-configuration; 引入了 commons-lang3, 以便使用其包含的一系列方便的工具类。

项目框架搭建完成后, 编写代码来实现服务“热配置”功能。

8.3.2 创建配置信息读取源

首先, 需要创建一个配置信息的读取源, 用于从 Consul-KV 上获取配置信息, 然后将这些信息存入内存。在创建读取源之前, 先定义两个常量, 以方便从 Consul-KV 读取信息。常量类 ServiceContents 代码如下:

```

package com.microservice.config.common;

public class ServiceContents {
    public static final String serviceName = "config";
    public static final String serviceTag = "dev";
}

```

在该类中, 定义了两个变量: 一个是 serviceName, 该值通常与 artifactId 相同; 另外一个 serviceTag, 该值用于区分环境, 通常开发环境使用“dev”, 线上环境使用“prod”。

使用该 tag 最大的好处是可以将 dev 和 prod 环境的配置分开，其实就是 Consul-KV 中的两个键。我们回到图 8-1，看一下这个 key-value 对中的 key 的组成，其实就是“service/{serviceName}/{serviceTag}”。当然，在实际项目中，我们不会把 serviceTag 写死在代码中，因为这样在上线的时候还需要改 tag，通常是在服务启动参数中进行指定，这里为了方便，笔者直接写死在了代码中。

准备工作做好之后，接下来创建配置信息读取源 ConsulConfigurationSource。代码如下：

```
package com.microservice.config.archaius;

import com.google.common.base.Optional;
import com.netflix.config.PollResult;
import com.netflix.config.PollableConfigurationSource;
import com.orbitz.consul.Consul;
import com.orbitz.consul.KeyValueClient;
import org.apache.commons.lang3.StringUtils;

import java.io.StringReader;
import java.util.HashMap;
import java.util.Map;
import java.util.Properties;

public class ConsulConfigurationSource implements PollableConfigurationSource {

    private String keyName;

    public ConsulConfigurationSource(String keyName) {
        this.keyName = keyName;
    }

    @Override
    public PollResult poll(boolean b, Object o) throws Exception {
        KeyValueClient kvClient = Consul.newClient().keyValueClient();
        Optional<String> kvOpt = kvClient.getValueAsString(keyName);
        String kvStr = StringUtils.EMPTY;
        if (kvOpt.isPresent()) {
            kvStr = kvOpt.get();
        }
    }
}
```

```

    Properties props = new Properties();
    props.load(new StringReader(kvStr)); //String->Properties

    Map<String, Object> propMap = new HashMap<>();
    for (Object key : props.keySet()) {
        propMap.put((String) key, props.get(key));
    }
    return PollResult.createFull(propMap);
}
}

```

该类实现了 `com.netflix.config.PolledConfigurationSource` 接口，并且实现了其中的 `poll()` 方法，在该方法中，首先根据传入的 `keyName` 从 Consul-KV 中获取配置信息，获取出来的配置信息是一个字符串，之后通过 `props.load(new StringReader(kvStr))` 将该字符串转换成 `Properties`，最后将该 `Properties` 中的 key-value 对组成 `Map` 元素，并通过该 `Map` 构建出 `PollResult` 对象。

其中，`keyName` 就是 Consul-KV 中的 K，在下边的 `ConsulPropertySourceInitializer` 类中会做指定。

另外，需要注意一点，该 `poll()` 方法默认每 60 s 执行一次，该值也可以由我们自己指定，同样在 `ConsulPropertySourceInitializer` 中指定。通过以固定的时间从 Consul-KV 中取出配置信息来实现“热配置”（动态获取配置信息而不需要重启服务）。

当然，也可以从另外的配置中心（例如，数据库）获取配置信息，只要重写这里的 `poll()` 方法即可。

8.3.3 实现服务启动时读取配置信息

为了实现服务启动时读取配置信息，我们需要清楚地知道 Spring Boot 的启动流程，先来看一下代码，然后再做分析。代码如下：

```

package com.microservice.config.archaius;

import com.microservice.config.common.ServiceConstants;
import com.netflix.config.*;
import org.springframework.context.ApplicationContextInitializer;
import org.springframework.context.ConfigurableApplicationContext;

```



```

public class ConsulPropertySourceInitializer implements
ApplicationContextInitializer<ConfigurableApplicationContext> {
    @Override
    public void initialize(ConfigurableApplicationContext
configurableApplicationContext) {
        System.setProperty("archaius.fixedDelayPollingScheduler.delayMills",
"10000");

        String keyName = "service/" + ServiceContants.serviceName + "/" +
ServiceContants.serviceTag;
        PolledConfigurationSource configSource = new ConsulConfigurationSource
(keyName);
        AbstractPollingScheduler scheduler = new FixedDelayPollingScheduler();
        DynamicConfiguration configuration = new
DynamicConfiguration(configSource, scheduler);
        ConfigurationManager.install(configuration);
    }
}

```

ConsulPropertySourceInitializer 类实现了 org.springframework.context.ApplicationContextInitializer 接口，并重写了其 initialize() 方法，这样做的原因是指定该 initialize 方法在 Spring Boot 启动过程的某个恰当时刻去执行，详细信息查看第 6 章相关内容。

在 initialize() 方法中，首先设置了 ConsulConfigurationSource 的 poll() 方法的执行间隔时间（即每隔多长时间从 Consul 拉取一次配置信息），这里设置为每隔 10 s，“archaius.fixedDelayPollingScheduler.delayMills”的默认值为 60000，即每隔 60 s 拉取一次配置信息。之后，组装了一个 keyName，图 8-1 中所示的 Consul-KV 中的 K 的格式实际上就是在这里指定的。接着创建了配置信息的读取源实例 configSource 以及用于启动读取数据源后台线程的 scheduler。然后，根据 configSource 和 scheduler 这两个信息来创建动态配置实例 configuration。最后，将该 configuration 实例安装到 ConfigurationManager 管理器中。

再将 ConsulPropertySourceInitializer 注册到 Spring Boot 的启动初始化器列表中，实现服务启动时读取配置信息的功能。该步骤在服务启动类中完成，代码如下：

```

package com.microservice.config;

import com.microservice.config.archaius.ConsulPropertySourceInitializer;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;

```



```
@SpringBootApplication
public class Application {
    public static void main(String[] args) {
        SpringApplication sa = new SpringApplication(Application.class);
        sa.addInitializers(new ConsulPropertySourceInitializer());
        sa.run(args);
    }
}
```

还是熟悉的启动类 `Application`，只是在执行其 `run()` 方法之前，为其添加了一个 `initializer` 实例，即 `ConsulPropertySourceInitializer` 实例。

8.3.4 动态获取配置信息

当通过 `Archaius` 定期从 `Consul-KV` 将配置信息读取到内存中后，我们就可以通过 `Archaius` 的相关 API 来动态获取配置信息了。获取配置信息的代码如下：

```
package com.microservice.config.controller;

import com.netflix.config.DynamicPropertyFactory;
import com.netflix.config.DynamicStringProperty;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;

@RestController
@RequestMapping("/archaius/test")
public class TestController {
    @RequestMapping("/getAnimalName")
    public String getAnimalName() {
        DynamicStringProperty dsp = DynamicPropertyFactory.getInstance().
            getStringProperty("animal.name", "cat");
        return dsp.get();
    }
}
```

在 `TestController` 中使用如下代码：

```
DynamicPropertyFactory.getInstance().getStringProperty("animal.name", "cat");
```

获取“animal.name”的值，如果没有读取到该值，使用默认值“cat”。

代码编写完成后，将代码编译、打包、复制到服务器，启动服务，之后在服务器上执行 `curl -X GET "http://localhost:8080/archaius/test/getAnimalName"`。

读取到配置文件后，再到 Consul-KV 上修改配置文件的值，再执行如上命令，看是否可以实现动态配置。

8.3.5 将配置信息动态加入 Spring 属性源的思路

在上一节中，通过 Archaius 的 API 实现了配置信息的动态获取，但是现在我们采用 Spring 的读取配置数据的方法，例如使用 `org.springframework.core.env.Environment` 读取配置信息已经不行了，因为配置信息只是存到了 Archaius 的 `PollResult` 中，而没有存入 Spring 的 `PropertySource` 中。想解决这个问题其实很简单，具体实现思路是：`ConsulPropertySource` 类实现 `org.springframework.core.env.MapPropertySource`，并将读取到的信息存储到 `MapPropertySource` 的顶级父接口 `org.springframework.core.env.PropertySource` 中的 `source` 变量中。最后，将 `ConsulPropertySource` 添加到 `MutablePropertySources` 的 `List<PropertySource<?>>` 中即可。笔者这里就不给出代码实现了。详细信息还是需要查看第 6 章相关内容，看完第 6 章的源码解析后，相信读者自己也可以实现将配置信息动态加入 Spring 属性源中。

8.4 再学一招：Archaius 关键源码解析

Archaius 的亮点就是配置信息的动态获取，或者称为“热配置”。下面就来看看 Archaius 实现“热配置”的原理。分为两部分：构造属性源；动态获取属性。

8.4.1 构造动态属性源

首先来看一下，构造属性源的正确方法：

```
public void initialize(ConfigurableApplicationContext
configurableApplicationContext) {
    System.setProperty("archaius.fixedDelayPollingScheduler.delayMills", "10000")
;

    String keyName = "service/" + ServiceContants.serviceName + "/" +
ServiceContants.serviceTag;
```

```

    PolledConfigurationSource configSource = new ConsulConfigurationSource
(keyName); //定义读取配置的源头
    AbstractPollingScheduler scheduler = new FixedDelayPollingScheduler(); //
设置读取配置文件的方法
    DynamicConfiguration configuration = new DynamicConfiguration(configSource,
scheduler);
    ConfigurationManager.install(configuration);
}

```

首先设置 `archaius.fixedDelayPollingScheduler.delayMills` 的值为 10000, 其实就是指定后边的定时任务每隔 10 s 执行一次; 然后创建 `PolledConfigurationSource` 接口的实现类 `ConsulConfigurationSource` 的实例; 再创建 `FixedDelayPollingScheduler` 实例, 看一下源码:

```

private int initialDelayMillis = 30000;
private int delayMillis = '\uea60';
public static final String INITIAL_DELAY_PROPERTY =
"archaius.fixedDelayPollingScheduler.initialDelayMills";
public static final String DELAY_PROPERTY =
"archaius.fixedDelayPollingScheduler.delayMills";

public FixedDelayPollingScheduler() {
    String initialDelayProperty =
System.getProperty("archaius.fixedDelayPollingScheduler.initialDelayMills");
    if(initialDelayProperty != null && initialDelayProperty.length() > 0) {
        this.initialDelayMillis = Integer.parseInt(initialDelayProperty);
    }

    String delayProperty =
System.getProperty("archaius.fixedDelayPollingScheduler.delayMills");
    if(delayProperty != null && delayProperty.length() > 0) {
        this.delayMillis = Integer.parseInt(delayProperty);
    }
}
}

```

该构造器为 `FixedDelayPollingScheduler` 的 `initialDelayMillis` 属性和 `delayMillis` 属性赋值。其中 `initialDelayMillis` 的默认值是 30s, 该值指定了定时任务创建完成 30s 后开始执行; `delayMillis` 默认值是 60s, 但是我们在前边指定为 10s, 就是说每隔 10s 执行一次定时任务。

执行完以上工作后，开始创建 `DynamicConfiguration` 实例，来看一下源码：

```
private AbstractPollingScheduler scheduler;
private PolledConfigurationSource source;

public DynamicConfiguration(PolledConfigurationSource source,
AbstractPollingScheduler scheduler) {
    this();
    this.startPolling(source, scheduler);
}

public DynamicConfiguration() {
}

public synchronized void startPolling(PolledConfigurationSource source,
AbstractPollingScheduler scheduler) {
    this.scheduler = scheduler;
    this.source = source;
    this.init(source, scheduler);
    scheduler.startPolling(source, this);
}

protected void init(PolledConfigurationSource source, AbstractPollingScheduler
scheduler) {
}
```

在 `DynamicConfiguration` 的构造器中调用了其 `startPolling` 方法，在该方法中调用了 `FixedDelayPollingScheduler` 的父类 `AbstractPollingScheduler` 的 `startPolling` 方法，源码如下：

```
public void startPolling(PolledConfigurationSource source, Configuration config)
{
    this.initialLoad(source, config);
    Runnable r = this.getPollingRunnable(source, config);
    this.schedule(r);
}
```

在该方法中，`AbstractPollingScheduler` 首先调用 `initialLoad` 方法，从 Consul 拉取配置信息并且将配置信息存在了 `PollResult` 的父类 `WatchedUpdateResult` 的 `Map<String, Object> complete` 属性中。之后创建一个定时任务来执行一个线程任务。下面来一步步地看下源码。

首先看 `AbstractPollingScheduler` 的 `initialLoad` 方法。

```
protected synchronized void initialLoad(final PolledConfigurationSource source,
final Configuration config) {
    PollResult result = null;
    try {
        result = source.poll(true, null);
        checkPoint = result.getCheckPoint();
        fireEvent(EventType.POLL_SUCCESS, result, null);
    } catch (Throwable e) {
        throw new RuntimeException("Unable to load Properties source from " +
source, e);
    }
    try {
        populateProperties(result, config);
    } catch (Throwable e) {
        throw new RuntimeException("Unable to load Properties", e);
    }
}
```

这里拉取配置信息并且将这些 key-value 对存储在 `PollResult` 的 `Map<String, Object> complete` 属性中，之后使用 `populateProperties` 方法将 `complete` 中的值添加或者更新到 `DynamicConfiguration` 的父类 `ConcurrentMapConfiguration` 的 `ConcurrentHashMap<String, Object> map` 属性中。该 `map` 属性也是程序中读取配置信息的地方。`populateProperties` 方法篇幅较大，不贴出来了。

再来看一下构造器的线程的 `getPollingRunnable` 方法：

```
protected Runnable getPollingRunnable(final PolledConfigurationSource source,
final Configuration config) {
    return new Runnable() {
        public void run() {
            log.debug("Polling started");
            PollResult result = null;
            try {
                result = source.poll(false, getNextCheckPoint(checkPoint));
                checkPoint = result.getCheckPoint();
                fireEvent(EventType.POLL_SUCCESS, result, null);
            } catch (Throwable e) {
                log.error("Error getting result from polling source", e);
            }
        }
    };
}
```



```

        fireEvent(EventType.POLL_FAILURE, null, e);
        return;
    }
    try {
        populateProperties(result, config);
    } catch (Throwable e) {
        log.error("Error occurred applying properties", e);
    }
}

};
}

```

该线程所做的工作其实与 `initialLoad` 方法几乎相同。

最后执行 `FixedDelayPollingScheduler` 的 `schedule` 方法，来看一下源代码：

```

@Override
protected synchronized void schedule(Runnable runnable) {
    executor = Executors.newScheduledThreadPool(1, new ThreadFactory() {
        @Override
        public Thread newThread(Runnable r) {
            Thread t = new Thread(r, "pollingConfigurationSource");
            t.setDaemon(true);
            return t;
        }
    });
    executor.scheduleWithFixedDelay(runnable, initialDelayMillis, delayMillis,
    TimeUnit.MILLISECONDS);
}

```

该方法启动了定时任务，其中的任务线程就是上边的线程。每隔 10 s 就会重新拉取配置信息，并且将其设置到 `ConcurrentHashMap<String, Object> map` 属性中，这样，程序获取配置信息时，获取的是新值。

这就是动态配置的原理！

8.4.2 动态获取属性

首先来看一下动态获取属性的正确方法：

```
DynamicStringProperty dsp = DynamicPropertyFactory.getInstance().getStringProperty  
("animal.name", "cat");  
  
return dsp.get();
```

该方法的实现源码比较曲折，代码线比较长，这里直接列出最后获取 `animal.name` 值的代码，即 `ConcurrentMapConfiguration` 类中的 `getProperty` 方法的代码。源码如下：

```
protected ConcurrentHashMap<String, Object> map;  
public Object getProperty(String key)  
{  
    return map.get(key);  
}
```

其中，`map` 属性就是前面所讲的属性值最终存取的地方，并随着定时任务的执行，`map` 中的值也在变。从而也就可以取出改变后的值，实现动态获取属性！

第 9 章

微服务进程间通信

9.1 常见的三种服务通信技术

常见的服务通信框架有很多，这里简单介绍三种：OkHttp、AsyncHttpClient 和 Retrofit。

OkHttp 是一种 http 客户端，其使用连接池技术来减少请求延迟，并且会对响应进行缓存以减少重复的网络请求，还无缝地支持 GZIP 以减少数据流量。OkHttp 的使用很简单，它的请求和响应 API 都是采用流式的 builder 模式来设计的；OkHttp 同时支持同步阻塞调用和异步调用（使用 callback 实现回调）。使用 OkHttp 至少需要 JDK1.7。

AsyncHttpClient 也是一种 http 客户端，设计该框架的目的是为了让 Java 应用可以很方便地执行 http 请求并且异步处理 http 响应。AsyncHttpClient 的底层是 Netty 框架（一个非常优秀的 rpc 框架），这就注定了它的效率是很高的，优于 OkHttp。使用最新版本的 AsyncHttpClient 需要 JDK1.8。

Retrofit 和 OkHttp 师出同门，都是 Square 公司开源的项目，Retrofit 实际上对 OkHttp 进行了封装。与 OkHttp 不同的是，Retrofit 将 url（或者更确切地说是 rest API）封装成了 Java 接口，在该接口上可以使用注解来指定一系列的相关信息，比如请求方式、请求参数等。我们不需要写该接口的实现类，Retrofit 会帮我们搞定；此外，Retrofit 还自己封装了 Gson，实现了将返回的 json 串自动转换为 POJO 的功能，当然我们可以实现自己的转换器，例如使用 Jackson，之后嵌入 Retrofit 中。在本章的“再学一招”部分，会对 Retrofit 的关键

源码进行解析。

本章会设计两个微服务：myserviceC 和 myserviceD。其中 myserviceD 提供一个简单的 http 接口，myserviceC 分别使用 OkHttp、AsyncHttpClient 和 Retrofit 来调用 myserviceD 提供的接口。下面首先开发 myserviceD 服务。

9.2 创建一个简单的被调用服务

9.2.1 搭建项目框架

myserviceD 的项目结构如图 9-1 所示。

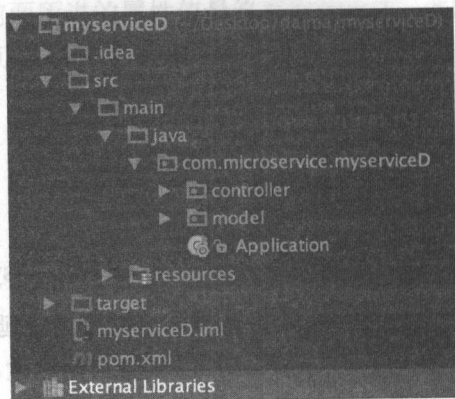


图9-1 myserviceD项目结构

其中，pom.xml 文件内容如下：

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/maven-v4_0_0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>com.microservice</groupId>
  <artifactId>myserviceD</artifactId>
  <version>1.0-SNAPSHOT</version>
```

```
<properties>
  <java.version>1.8</java.version>
</properties>
<parent>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-parent</artifactId>
  <version>1.4.3.RELEASE</version>
</parent>

<dependencies>
  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
  </dependency>
  <dependency>
    <groupId>io.springfox</groupId>
    <artifactId>springfox-swagger2</artifactId>
    <version>2.2.2</version>
  </dependency>
  <dependency>
    <groupId>io.springfox</groupId>
    <artifactId>springfox-swagger-ui</artifactId>
    <version>2.2.2</version>
  </dependency>
  <dependency>
    <groupId>org.projectlombok</groupId>
    <artifactId>lombok</artifactId>
    <version>1.16.8</version>
  </dependency>
</dependencies>

<build>
  <plugins>
    <plugin>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-maven-plugin</artifactId>
    </plugin>
  </plugins>
</build>
</project>
```


引入 spring-boot-starter-web、springfox-swagger2、springfox-swagger-ui 及 Lombok 4 个依赖。

项目框架搭建完成之后，编写一个简单的被调用接口。

9.2.2 实现一个简单的被调用接口

首先编写一个启动类，代码如下：

```
package com.microservice.myserviceD;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import springfox.documentation.swagger2.annotations.EnableSwagger2;

@SpringBootApplication
@EnableSwagger2
public class Application {
    public static void main(String[] args) {
        SpringApplication sa = new SpringApplication(Application.class);
        sa.run(args);
    }
}
```

还是熟悉的服务启动类。

然后编写一个模型类作为接口返回值。模型类代码如下：

```
package com.microservice.myserviceD.model;

import lombok.AllArgsConstructor;
import lombok.Getter;

@Getter
@AllArgsConstructor
public class User {
    private String username;
    private int age;
}
```

使用 `@Getter` 注解生成 `getter` 方法，在 Jackson 转换时需要用到它；使用 `@AllArgsConstructor` 注解添加全参构造器，以便构造并初始化对象。

最后编写被调用接口，代码如下：

```
package com.microservice.myserviceD.controller;

import com.microservice.myserviceD.model.User;
import io.swagger.annotations.Api;
import io.swagger.annotations.ApiOperation;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestMethod;
import org.springframework.web.bind.annotation.RequestParam;
import org.springframework.web.bind.annotation.RestController;

@Api("myserviceD 相关 API")
@RestController
@RequestMapping("/myserviceD")
public class MyserviceDController {

    @ApiOperation("根据用户名获取用户信息")
    @RequestMapping(value = "/getUser", method = RequestMethod.GET)
    public User getUser(@RequestParam("username") String username) {
        User user = null;
        if (username.equals("小娜")) {
            user = new User(username, 18);
        } else {
            user = new User("小刚", 20);
        }
        return user;
    }
}
```

该类只提供了一个接口，其根据用户名返回不同的 User 对象实例。

myserviceD 的代码很简单，就是一个简单的 SpringBoot 应用。之后，分别使用 `OkHttp`、`AsyncHttpClient` 和 `Retrofit` 进行服务调用。

9.3 使用 OkHttp 实现服务通信

9.3.1 搭建项目框架

myserviceC 的项目结构如图 9-2 所示。

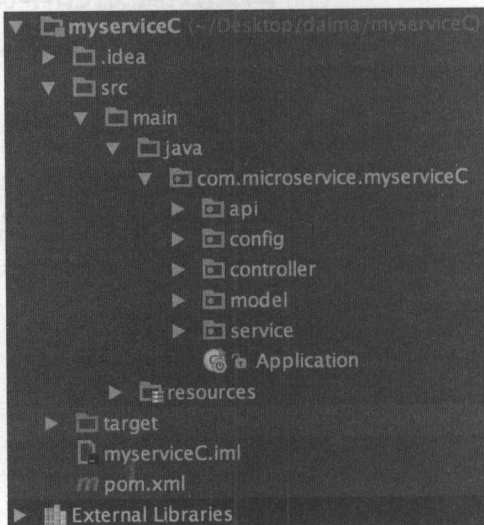


图9-2 myserviceC项目结构

在其 pom.xml 文件中引入如下依赖：

```
<dependency>
    <groupId>com.alibaba</groupId>
    <artifactId>fastjson</artifactId>
    <version>1.1.15</version>
</dependency>
<!-- OkHttp -->
<dependency>
    <groupId>com.squareup.okhttp</groupId>
    <artifactId>okhttp</artifactId>
    <version>2.7.5</version>
</dependency>
```

引入 OkHttp 2.7.5 和 fastjson，其他需要引入的依赖就是 myserviceD 引入的那些依赖，这里就不再赘述了。

项目框架搭建完成后，需要编写一个 OkHttp 调用实体类，为以后的服务通信做准备。

9.3.2 创建 OkHttp 调用实体类

首先，编写一个服务启动类，myserviceC 的服务启动类也与 myserviceD 的相同，不再赘述。下面创建一个调用实体类，代码如下：

```
package com.microservice.myserviceC.config;

import com.squareup.okhttp.OkHttpClient;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;

import java.util.concurrent.TimeUnit;

@Configuration
public class MyserviceCConfig {
    @Bean
    public OkHttpClient okHttpClient() {
        OkHttpClient okHttpClient = new OkHttpClient();
        okHttpClient.setConnectTimeout(2000, TimeUnit.MILLISECONDS); // 连接超时时间, 默认是 2s
        okHttpClient.setReadTimeout(3000, TimeUnit.MILLISECONDS); // 读取超时时间, 默认是 3s
        okHttpClient.setWriteTimeout(3000, TimeUnit.MILLISECONDS); // 写入超时时间, 默认是 3s
        return okHttpClient;
    }
}
```

在 MyserviceCConfig 类中，构造了一个 com.squareup.okhttp.OkHttpClient 单例，并指定连接超时时间是 2 s，读取超时时间是 3 s，写入超时时间是 3 s，在实际开发中，这些配置参数最好配置在配置中心中，方便后续修改。值得一提的是，官方推荐全局使用一个 OkHttpClient，所以笔者在这里构建了一个单例 Bean。

这里解释一下三个易混的超时概念。

- 连接超时：connectTimeout，A 服务建立与 B 服务的连接的时间，即完成三次握手的最大时间。

- 读取超时：A 服务接收到下一个字节与上一个字节之间的最大时间，即如果在接收到来自于 B 服务的上一个字节后，经过读取超时时间后，还没有接收到来自于 B 服务的下一个字节，则抛出超时异常。
- 写入超时：从 A 服务发出一个报文到接收到来自于 B 服务的 ack 消息的时间，通常不设置该参数。

创建好 OkHttpClient 实例之后，就可以使用该实例来进行服务调用了。

9.3.3 实现服务通信功能

首先，创建一个模型类，接收 myserviceD 被调用接口的返回值，代码如下：

```
package com.microservice.myserviceC.model;

import lombok.Getter;
import lombok.Setter;

@Setter
@Getter
public class User {
    private String username;
    private int age;
}
```

之后，编写服务调用代码。代码如下：

```
package com.microservice.myserviceC.service;

import com.alibaba.fastjson.JSON;
import com.microservice.myserviceC.model.User;
import com.squareup.okhttp.OkHttpClient;
import com.squareup.okhttp.Request;
import com.squareup.okhttp.Response;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;

import java.io.IOException;

@Service
```



```

public class MyServiceCService {
    @Autowired
    private OkHttpClient okHttpClient;

    public User getUserByOkHttp(String username) {
        User user = null;
        Response response = null;
        String url = "http://localhost:8080/myserviceD/getUser?username=" + username;
        try {
            Request request = new Request.Builder().url(url).build();
            response = okHttpClient.newCall(request).execute();
            String userStr = response.body().string();
            user = JSON.parseObject(userStr, User.class);
        } catch (IOException e) {
            e.printStackTrace();
        } finally {
            if (response != null && response.body() != null) {
                try {
                    response.body().close(); // 一定要关闭，不然会泄露资源
                } catch (IOException e) {
                    e.printStackTrace();
                }
            }
        }
        return user;
    }
}

```

在 `getUserByOkHttp` 方法中，首先组装了访问 `myserviceD` 中的接口的 `url`，然后构建了请求实例 `request`，之后使用之前实例化好的 `okHttpClient` 执行调用并同步接收返回值，最后使用 `fastjson` 将返回值反序列化成 `User` 实例。最后的最后，一定要关闭 `body` 体!!!

下面，编写 `controller` 进行调用。代码如下：

```

package com.microservice.myserviceC.controller;

import com.microservice.myserviceC.service.MyServiceCService;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestMethod;

```

```
import org.springframework.web.bind.annotation.RequestParam;
import org.springframework.web.bind.annotation.RestController;

import com.microservice.myServiceC.model.User;

import io.swagger.annotations.Api;
import io.swagger.annotations.ApiOperation;

@Api("myServiceC 相关 API")
@RestController
@RequestMapping("/myServiceC")
public class MyServiceCController {
    @Autowired
    private MyServiceCService myServiceCService;

    @ApiOperation("使用 okHttp 调用其他服务")
    @RequestMapping(value = "/okhttp/getUser", method = RequestMethod.GET)
    public User getUserByOkHttp(@RequestParam("username") String username) {
        return myServiceCService.getUserByOkHttp(username);
    }
}
```

运行服务，然后使用 Swagger 进行测试。至此，我们就完成了使用 OkHttp 实现服务通信的功能！

9.3.4 Spring Boot 指定服务启动端口的三种方式

在实际开发中，同一个机器上如果部署两个服务，则常常会出现 8080 端口已经被其他服务占用的情况，这个时候，另一个服务就无法再使用 8080 端口了。例如这里的 myServiceD 已经占用了 8080 端口，myServiceC 就需要使用其他端口。指定服务端口共有三种方式：

- 设置 VM 启动参数，如图 9-3 所示。

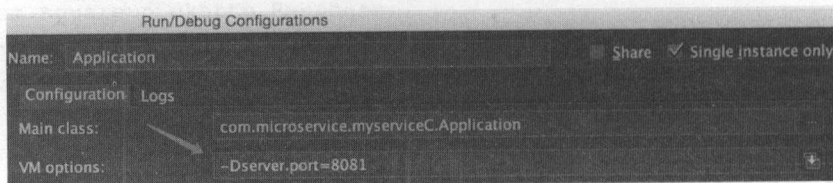


图9-3 指定端口配置

- 在配置文件中添加: `server.port=8081`。
- 第三种方式: 启动类实现 `EmbeddedServletContainerCustomizer` 接口, 重写其方法, 添加 `container.setPort(8081)`。

下面, 我们来看一下使用 `AsyncHttpClient` 实现服务通信的代码实现。

9.4 使用 AsyncHttpClient 实现服务通信

我们在 `myserviceC` 的基础上完成代码开发。

9.4.1 搭建项目框架

在 `pom.xml` 文件中引入 `AsyncHttpClient` 的依赖, 如下:

```
<!-- async-http-client -->
<dependency>
    <groupId>com.ning</groupId>
    <artifactId>async-http-client</artifactId>
    <version>1.9.31</version>
</dependency>
```

项目框架搭建完成后, 需要编写一个 `AsyncHttpClient` 调用实体类, 为以后的服务通信做准备。

9.4.2 创建 AsyncHttpClient 调用实体类

`AsyncHttpClient` 调用实体类的关键代码如下:

```
@Bean
public AsyncHttpClient asyncHttpClient() {
    AsyncHttpClientConfig.Builder builder = new AsyncHttpClientConfig.
Builder()
        .setRequestTimeout(5000)
        .setConnectTimeout(2000)
        .setReadTimeout(3000);

    return new AsyncHttpClient(builder.build());
}
```

该方法创建了一个 `com.ning.http.client.AsyncHttpClient` 实例，并指定了连接超时时间为 2 s，读取超时时间为 3 s，请求的超时时间为 5 s。

9.4.3 实现服务通信功能

创建好 `AsyncHttpClient` 实例后，就可以使用该实例进行服务调用了。服务调用代码如下：

```
@Autowired
private AsyncHttpClient asyncHttpClient;

public User getUserByAsyncHttpClient(String username) {
    User user = null;
    String url = "http://localhost:8080/myserviceD/getUser?username=" +
username;
    com.ning.http.client.Request request = new
RequestBuilder().setUrl(url).build();
    Future<com.ning.http.client.Response> responseFuture =
asyncHttpClient.executeRequest(request);
    try {
        com.ning.http.client.Response response = responseFuture.get();
        String reponseStr = response.getResponseBody();
        user = JSON.parseObject(reponseStr, User.class);
    } catch (InterruptedException e) {
        e.printStackTrace();
    } catch (ExecutionException e) {
        e.printStackTrace();
    } catch (IOException e) {
        e.printStackTrace();
    }
    return user;
}
```

在 `getUserByAsyncHttpClient` 方法中，首先组装了访问 `myserviceD` 中的接口的 url，之后构建了请求实例 `request`，然后使用之前实例化好的 `asyncHttpClient` 执行调用并使用 `java.util.concurrent.Future` 实现异步接收返回值，最后使用 `fastjson` 将返回值反序列化成 `User` 实例。这里有一个需要注意的地方，就是在实际开发中，对于异常，不要直接使用 `e.printStackTrace()`，通常，可以将异常的堆栈信息放在日志中。

下面，编写 controller 进行调用。代码如下：

```
@ApiOperation("使用 asyncHttpClient 调用其他服务")
@RequestMapping(value = "/asyncHttpClient/getUser", method = RequestMethod.GET)
public User getUserByAsyncHttpClient(@RequestParam("username") String username) {
    return myserviceCService.getUserByAsyncHttpClient(username);
}
```

运行服务，进入 Swagger 进行测试。至此，我们就完成了使用 AsyncHttpClient 实现服务通信的功能！下面，我们来看一下使用 Retrofit 实现服务通信的代码实现。

9.5 使用 Retrofit 实现服务通信

还是在 myserviceC 的基础上完成代码开发。

9.5.1 搭建项目框架

在 pom.xml 文件中引入 Retrofit 的依赖，如下：

```
<!-- retrofit -->
<dependency>
    <groupId>com.squareup.retrofit</groupId>
    <artifactId>retrofit</artifactId>
    <version>1.9.0</version>
</dependency>
```

引入 Retrofit1.9.0 的 jar 包，这是目前常用的版本，Retrofit2.x 做了很大的更新。

项目框架搭建完成后，编写使用 Retrofit 进行服务通信的代码。

9.5.2 创建调用接口并实例化接口

要想使用 Retrofit 进行服务通信，首先需要将被调用接口接口化。这里笔者将 myserviceD 的被调用接口接口化，代码如下：

```
package com.microservice.myserviceC.api;

import com.microservice.myserviceC.model.User;
import retrofit.http.GET;
```



```
import retrofit.http.Query;

public interface MyserviceDApi {

    @GET("/myserviceD/getUser")
    User getUser(@Query("username") String username);

}
```

该接口指定了调用远程服务 `myserviceD` 的方法 `getUser` 的基本路径 `uri`、参数、返回值和请求方式，有以下几点需要注意：

1. 在调用时，该基本路径 `uri` 会与之后的 `RestAdapter` 的 `endpoint` 组成完整的访问路径以被调用者访问。
2. 请求方式比较常用的有：`@GET`、`@POST`、`@PUT` 和 `@DELETE` 等，分别对应于 `restful` 风格中的查询、增加、修改和删除。
3. 入参使用 `@Query` 传递 `query` 域中的值，Spring MVC 使用 `@RequestParam` 接收该域中的值；入参使用 `@Path` 传递 `path` 域中的值，Spring MVC 使用 `@PathVariable` 接收该域中的值；入参使用 `@Header` 传递 `header` 域中的值，Spring MVC 使用 `@RequestHeader` 接收该域中的值；入参使用 `@Body` 传递 `body` 域中的值（通常是一个 `pojo` 对象），Spring MVC 使用 `@RequestBody` 接收该域中的值。

将被调用接口接口化之后，还需要将该接口“实例化”，代码如下：

```
@Bean
public MyserviceDApi myserviceDApi() {
    RestAdapter restAdapter = new
RestAdapter.Builder().setEndpoint("http://localhost:8080/").build();
    MyserviceDApi myserviceDApi =restAdapter.create(MyserviceDApi.class);
    return myserviceDApi;
}
```

在该方法中，首先创建了一个 `RestAdapter` 实例 `restAdapter`，并指定了 `endpoint`，最后使用 `restAdapter` 创建出一个 `MyserviceDApi` 实例（值得注意的是，为什么 `MyserviceDApi` 明明是一个接口，却可以构建实例？答案是动态代理，参考本章的“再学一招”部分），供调用方使用。在创建 `RestAdapter` 实例时，我们可以自己指定 `json` 转换器，自己指定错误处理器（需要实现 `retrofit.ErrorHandler`），自己指定 `Client`（需要实现 `retrofit.client.Client`）。

在 Retrofit 发起调用时,可以将此调用委托给该 Client,该 Client 再去进行真正的服务调用。

9.5.3 实现服务通信功能

一切准备就绪之后,编写代码实现服务通信。代码如下:

```
@Autowired
private MyServiceDApi myServiceDApi;

public User getUserByRetrofit(String username) {
    return myServiceDApi.getUser(username);
}
```

这里实现了对远程方法的调用,调用远程方法就和调用本地方法一样简单,一样明了。

最后,编写 controller。代码如下:

```
@ApiOperation("使用 Retrofit 调用其他服务")
@RequestMapping(value = "/retrofit/getUser", method = RequestMethod.GET)
public User getUserByRetrofit(@RequestParam("username") String username) {
    return myServiceCService.getUserByRetrofit(username);
}
```

运行服务,使用 Swagger 进行测试。至此,我们就完成了使用 Retrofit 实现服务通信的功能!

最后说一句,虽然 OkHttp 使用起来比较简单,但是笔者使用较少,因为论性能,AsyncHttpClient 优于 OkHttp,论设计,Retrofit 优于 OkHttp,所以笔者更推荐使用后两种技术。但是当引入服务路由之后,AsyncHttpClient 使用起来依然简单,但是 Retrofit 就要使用自定义的 retrofit.client.Client 的实现类了,当然编写实现类也很简单,完全可以模仿 Retrofit 源码中给出的一些样例。

9.6 再学一招: Retrofit 源码解析

Retrofit 源码主要有两部分:构造 RestAdapter;构造请求方法的接口类(这里是 MyServiceDApi);进行请求调用。

9.6.1 构造 RestAdapter

来看一下正确地构造 RestAdapter 的方法：

```
RestAdapter restAdapter = new RestAdapter.Builder().setEndpoint("http://localhost:8080/").build();
```

这是典型的构造器模式，在 RestAdapter 中含有一个内部类 Builder。由于该类的代码比较长，这里只列出部分会用到的代码，源码如下：

```
public static class Builder {
    private Endpoint endpoint;
    private Client.Provider clientProvider;
    private Executor httpExecutor;
    private Executor callbackExecutor;
    private RequestInterceptor requestInterceptor;
    private Converter converter;
    private Profiler profiler;
    private ErrorHandler errorHandler;

    public Builder setEndpoint(String endpoint) {
        if (endpoint == null || endpoint.trim().length() == 0) {
            throw new NullPointerException("Endpoint may not be blank.");
        }
        this.endpoint = Endpoints.newFixedEndpoint(endpoint);
        return this;
    }

    public Builder setClient(final Client client) {
        if (client == null) {
            throw new NullPointerException("Client may not be null.");
        }
        return setClient(new Client.Provider() {
            @Override public Client get() {
                return client;
            }
        });
    }

    public Builder setClient(Client.Provider clientProvider) {
```

```
    if (clientProvider == null) {
        throw new NullPointerException("Client provider may not be null.");
    }
    this.clientProvider = clientProvider;
    return this;
}

public Builder setConverter(Converter converter) {
    if (converter == null) {
        throw new NullPointerException("Converter may not be null.");
    }
    this.converter = converter;
    return this;
}

public Builder setErrorHandler(ErrorHandler errorHandler) {
    if (errorHandler == null) {
        throw new NullPointerException("Error handler may not be null.");
    }
    this.errorHandler = errorHandler;
    return this;
}

public RestAdapter build() {
    if (endpoint == null) {
        throw new IllegalArgumentException("Endpoint may not be null.");
    }
    ensureSaneDefaults();
    return new RestAdapter(endpoint, clientProvider, httpExecutor, callbackExecutor,
        requestInterceptor, converter, profiler, errorHandler, log, logLevel);
}

private void ensureSaneDefaults() {
    if (converter == null) {
        converter = Platform.get().defaultConverter();
    }
    if (clientProvider == null) {
        clientProvider = Platform.get().defaultClient();
    }
    if (httpExecutor == null) {

```



```

    httpExecutor = Platform.get().defaultHttpExecutor();
}
if (callbackExecutor == null) {
    callbackExecutor = Platform.get().defaultCallbackExecutor();
}
if (errorHandler == null) {
    errorHandler = ErrorHandler.DEFAULT;
}
if (log == null) {
    log = Platform.get().defaultLog();
}
if (requestInterceptor == null) {
    requestInterceptor = RequestInterceptor.NONE;
}
}
}

```

`setEndpoint` 方法用于设定访问的基本 URL，例如，`http://localhost:8080/`，该 URL 会与之后的接口中的 URI 拼接；`setClient` 方法用于指定底层的 http 调用引擎，例如，Retrofit 自己使用的是 `OkHttp`，我们可以通过 `AsyncHttpClient` 实现一个自己的底层调用引擎；`setConverter` 方法用于指定进行序列化和反序列化的转换器，默认使用 `GsonConverter`，我们可以使用 `JacksonConverter` 来实现；`setErrorHandler` 用于指定出错时的错误处理器。以上是我们最常定制的 4 个部分，这里笔者只指定了 `Endpoint`。后面看一下 `build()` 方法。在该方法中，首先要求 `Endpoint` 不能为空，然后初始化一系列属性，最后创建了 `RestAdapter` 对象。

9.6.2 初始化 `RestAdapter.Builder` 属性

在初始化各个属性的 `ensureSaneDefaults()` 方法中，首先获取 `platform`，之后获取 `platform` 中的各个属性的默认值。来看一下获取 `platform` 的源码：

```

private static final Platform PLATFORM = findPlatform();

static Platform get() {
    return PLATFORM;
}

private static Platform findPlatform() {

```



```

try {
    Class.forName("android.os.Build");
    if (Build.VERSION.SDK_INT != 0) {
        return new Android();
    }
} catch (ClassNotFoundException ignored) {
}

if (System.getProperty("com.google.appengine.runtime.version") != null) {
    return new AppEngine();
}

return new Base();
}

```

如果有名为“android.os.Build”的 Class，则创建一个 android 平台；否则，如果系统属性中设置了“com.google.appengine.runtime.version”，则创建一个 AppEngine；否则，创建一个 Base 平台。我们这里创建了 Base 平台。源码如下：

```

private static class Base extends Platform {
    @Override Converter defaultConverter() {
        return new GsonConverter(new Gson());
    }

    @Override Client.Provider defaultClient() {
        final Client client;
        if (hasOkHttpOnClasspath()) {
            client = OkHttpClientInstantiator.instantiate();
        } else {
            client = new UrlConnectionClient();
        }
        return new Client.Provider() {
            @Override public Client get() {
                return client;
            }
        };
    }

    @Override Executor defaultHttpExecutor() {
        return Executors.newCachedThreadPool(new ThreadFactory() {

```

```

@Override public Thread newThread(final Runnable r) {
    return new Thread(new Runnable() {
        @Override public void run() {
            Thread.currentThread().setPriority(MIN_PRIORITY);
            r.run();
        }
    }, RestAdapter.IDLE_THREAD_NAME);
}

});

@Override Executor defaultCallbackExecutor() {
    return new Utils.SynchronousExecutor();
}

@Override RestAdapter.Log defaultLog() {
    return new RestAdapter.Log() {
        @Override public void log(String message) {
            System.out.println(message);
        }
    };
}
}
}

```

该平台指定了转换器为 `GsonConverter`；而具体使用哪一个 `http` 通信框架，首先会在类路径下寻找是否有 `“com.squareup.okhttp.OkHttpClient”` 类，如果有，则使用 `OkHttp`；否则使用 `URLConnectionClient`。来看一下相关的源码，首先看一下 `hasOkHttpOnClasspath()` 方法：

```

private static boolean hasOkHttpOnClasspath() {
    try {
        Class.forName("com.squareup.okhttp.OkHttpClient");
        return true;
    } catch (ClassNotFoundException ignored) {
    }
    return false;
}

```

再看一下 `OkClientInstantiator.instantiate()` 的源码：

```

private static class OkClientInstantiator {

```

```
static Client instantiate() {  
    return new OkHttpClient();  
}  
}
```

看一下创建 OkHttpClient 的源码:

```
public OkHttpClient() {  
    this(generateDefaultOkHttp());  
}  
  
private static OkHttpClient generateDefaultOkHttp() {  
    OkHttpClient client = new OkHttpClient();  
    client.setConnectTimeout(Defaults.CONNECT_TIMEOUT_MILLIS,  
TimeUnit.MILLISECONDS);  
    client.setReadTimeout(Defaults.READ_TIMEOUT_MILLIS,  
TimeUnit.MILLISECONDS);  
    return client;  
}  
  
public OkHttpClient(OkHttpClient client) {  
    if (client == null) throw new NullPointerException("client == null");  
    this.client = client;  
}
```

其实就是创建了一个 OkHttpClient 实例,并且指定了 connectTimeout 为 15 s, readTimeout 为 20 s。

OkClient 类是一个非常重要的类,后续真正的 http 调用就是在该类中进行的。

如果类路径下没有“com.squareup.okhttp.OkHttpClient”类,则使用 UrlConnectionClient。该类使用的底层技术是 java.net.HttpURLConnection。通常,我们会使用 OkHttpClient,不会使用太原始的 UrlConnectionClient。

最后,值得注意的一点是,OkHttp 实现了 retrofit.client.Client 接口,这也是我们实现自定义的 Client 的方式。这是第一个模仿点!!!

9.6.3 创建 RestAdapter 实例

设置完属性之后,就可以根据这些属性创建 RestAdapter 实例了。来看一下源码:

```

final Endpoint server;
private final Client.Provider clientProvider;
final Executor httpExecutor;
final Executor callbackExecutor;
final RequestInterceptor requestInterceptor;
final Converter converter;
private final Profiler profiler;
final ErrorHandler errorHandler;
final Log log;
volatile LogLevel logLevel;

private RestAdapter(Endpoint server, Client.Provider clientProvider, Executor
httpExecutor,
    Executor callbackExecutor, RequestInterceptor requestInterceptor,
Converter converter,
    Profiler profiler, ErrorHandler errorHandler, Log log, LogLevel logLevel)
{
    this.server = server;
    this.clientProvider = clientProvider;
    this.httpExecutor = httpExecutor;
    this.callbackExecutor = callbackExecutor;
    this.requestInterceptor = requestInterceptor;
    this.converter = converter;
    this.profiler = profiler;
    this.errorHandler = errorHandler;
    this.log = log;
    this.logLevel = logLevel;
}

```

实际上就是先创建 `RestAdapter` 实例，然后将内部类 `Builder` 的属性设置给外部类 `RestAdapter`。

9.6.4 构造请求方法的接口类

使用如下：

```
MyServiceDApi myServiceDApi = restAdapter.create(MyServiceDApi.class);
```

`MyServiceDApi` 的代码如下：

```
package com.microservice.myServiceC.api;

import com.microservice.myServiceC.model.User;
import retrofit.http.GET;
import retrofit.http.Query;

public interface MyServiceDApi {
    @GET("/myServiceD/getUser")
    User getUser(@Query("username") String username);
}
```

来看一下 `restAdapter.create(MyServiceDApi.class)` 的源码:

```
public <T> T create(Class<T> service) {
    Utils.validateServiceClass(service);
    return (T) Proxy.newProxyInstance(service.getClassLoader(), new Class<?>[] {
        service },
        new RestHandler(getMethodInfoCache(service)));
}
```

就两步，第一步校验输入的 `service` 的合法性；第二步使用动态代理创建对象。

9.6.5 校验 service 接口的合法性

先来看一下校验源码:

```
static <T> void validateServiceClass(Class<T> service) {
    if (!service.isInterface()) {
        throw new IllegalArgumentException("Only interface endpoint definitions
are supported.");
    }
    if (service.getInterfaces().length > 0) {
        throw new IllegalArgumentException("Interface definitions must not extend
other interfaces.");
    }
}
```

传入的 `service` 必须是接口而且该接口不能扩展其他接口。

9.6.6 使用动态代理创建对象

使用动态代理创建对象，首先构造入参：

```
private final Map<Class<?>, Map<Method, RestMethodInfo>> serviceMethodInfoCache =
    new LinkedHashMap<Class<?>, Map<Method, RestMethodInfo>>();

Map<Method, RestMethodInfo> getMethodInfoCache(Class<?> service) {
    synchronized (serviceMethodInfoCache) {
        Map<Method, RestMethodInfo> methodInfoCache = serviceMethodInfoCache.get(
            service);
        if (methodInfoCache == null) {
            methodInfoCache = new LinkedHashMap<Method, RestMethodInfo>();
            serviceMethodInfoCache.put(service, methodInfoCache);
        }
        return methodInfoCache;
    }
}
```

`serviceMethodInfoCache` 用于存放 `service` 接口中的各个方法信息。此处初始化了一个空的 `LinkedHashMap`。

```
private final Map<Method, RestMethodInfo> methodDetailsCache;

RestHandler(Map<Method, RestMethodInfo> methodDetailsCache) {
    this.methodDetailsCache = methodDetailsCache;
}
```

其中，`RestHandler` 实现了接口 `java.lang.reflect.InvocationHandler`，这是动态代理使用真实对象调用方法的地方。构造好参数之后，使用 `Proxy.newProxyInstance` 方法创建对象。这是 Java 的基础，不做解析了。到此为止，`MyServiceDApi` 实例就创建成功了！

9.6.7 进行请求调用

如下：

```
@Autowired
private MyServiceDApi myServiceDApi;

public User getUserByRetrofit(String username) {
```

```

    return myserviceDApi.getUser(username);
}

```

为了清晰，列出 MyserviceDApi 的 getUser(String username)方法。如下：

```

@GET("/myserviceD/getUser")
User getUser(@Query("username") String username);

```

当我们调用这个方法时，实际上是 RestHandler 使用其 invoke 方法进行调用。这里只分析同步调用的方式，列出代码中比较重要的部分，如下：

```

private final Map<Method, RestMethodInfo> methodDetailsCache;

@Override public Object invoke(Object proxy, Method method, final Object[] args)
    throws Throwable {

    ...

    final RestMethodInfo methodInfo = getMethodInfo(methodDetailsCache, method);

    if (methodInfo.isSynchronous) {
        try {
            return invokeRequest(requestInterceptor, methodInfo, args);
        } catch (RetrofitError error) {
            Throwable newError = errorHandler.handleError(error);
            if (newError == null) {
                throw new IllegalStateException("Error handler returned null for
wrapped exception.",
                    error);
            }
            throw newError;
        }
    }

    ...

    return null;
}

```

该方法的入参 method 携带了方法的 name、返回值、参数值信息；args 是传入的参数值。在该方法中首先获取 RestMethodInfo，然后进行方法调用。

9.6.8 获取 RestMethodInfo 实例

源码如下：

```
static RestMethodInfo getMethodInfo (Map<Method, RestMethodInfo> cache, Method
method) {
    synchronized (cache) {
        RestMethodInfo methodInfo = cache.get(method);
        if (methodInfo == null) {
            methodInfo = new RestMethodInfo(method);
            cache.put(method, methodInfo);
        }
        return methodInfo;
    }
}
```

先从 `methodDetailsCache` 中获取 `RestMethodInfo` 信息,如果缓存中没有,则通过 `method` 信息新建 `RestMethodInfo` 实例,之后将该实例存入 `methodDetailsCache`,最后返回该实例。该 `RestMethodInfo` 实例中存储了一个方法(例如, `getUser` 方法)所有的调用信息,包括是否是同步调用、返回值类型、请求方法、请求 URL 等。只是此时还未初始化关键的信息。

9.6.9 进行方法调用

来看一下 `invokeRequest` 方法的源码,方法很长,列出关键源码:

```
private Object invokeRequest(RequestInterceptor requestInterceptor,
RestMethodInfo methodInfo,
    Object[] args) {
    String url = null;
    try {
        methodInfo.init();
        String serverUrl = server.getUrl();
        RequestBuilder requestBuilder = new RequestBuilder(serverUrl, methodInfo,
converter);
        requestBuilder.setArguments(args);

        requestInterceptor.intercept(requestBuilder);

        Request request = requestBuilder.build();
```

```
url = request.getUrl();

...

long start = System.nanoTime();
Response response = clientProvider.get().execute(request);
long elapsedTime = TimeUnit.NANOSECONDS.toMillis(System.nanoTime() - start);
int statusCode = response.getStatus();
Type type = methodInfo.responseObjectType;

if (statusCode >= 200 && statusCode < 300) { // 2XX == successful request
    if (type.equals(Response.class)) {
        if (!methodInfo.isStreaming) {
            response = Utils.readBodyToBytesIfNecessary(response);
        }

        if (methodInfo.isSynchronous) {
            return response;
        }
        return new ResponseWrapper(response, response);
    }

    TypedInput body = response.getBody();
    if (body == null) {
        if (methodInfo.isSynchronous) {
            return null;
        }
        return new ResponseWrapper(response, null);
    }

    ExceptionCatchingTypedInput wrapped = new ExceptionCatchingTypedInput
(body);
    try {
        Object convert = converter.fromBody(wrapped, type);
        logResponseBody(body, convert);
        if (methodInfo.isSynchronous) {
            return convert;
        }
        return new ResponseWrapper(response, convert);
    } catch (ConversionException e) {
```



```

        ...
        //抛出异常, 结束流程
    }
}

response = Utils.readBodyToBytesIfNecessary(response);
throw RetrofitError.httpError(url, response, converter, type);
}

...

}
}

```

该方法一共执行了如下几步：加载上述 `RestMethodInfo` 的剩余属性；构建请求体 `retrofit.client.Request`；调用 `clientProvider` 进行真正的调用；处理响应。

9.6.10 加载 `RestMethodInfo` 的剩余属性

源码如下：

```

synchronized void init() {
    if (loaded) return;

    parseMethodAnnotations();
    parseParameters();

    loaded = true;
}

```

这几行代码一共做了两件事：解析方法上的注解；解析入参注解。一个 `Retrofit` 接口 `API` 只包含这两种注解。首先看一下解析方法上的注解的形式，源码如下：

```

private void parseMethodAnnotations() {
    for (Annotation methodAnnotation : method.getAnnotations()) {
        Class<? extends Annotation> annotationType =
            methodAnnotation.annotationType();
        RestMethod methodInfo = null;

        for (Annotation innerAnnotation : annotationType.getAnnotations()) {

```



```

    if (RestMethod.class == innerAnnotation.annotationType()) {
        methodInfo = (RestMethod) innerAnnotation;
        break;
    }
}

if (methodInfo != null) {
    if (requestMethod != null) {
        throw methodError("Only one HTTP method is allowed. Found: %s and %s.",
            requestMethod,
            methodInfo.value());
    }
    String path;
    try {
        path = (String) annotationType.getMethod("value").invoke(methodAnnotation);
    } catch (Exception e) {
        throw methodError("Failed to extract String 'value' from %s annotation.",
            annotationType.getSimpleName());
    }
    parsePath(path);
    requestMethod = methodInfo.value();
    requestHasBody = methodInfo.hasBody();
}
...
}

if (requestMethod == null) {
    throw methodError("HTTP method annotation is required (e.g., @GET, @POST,
etc.).");
}

if (!requestHasBody) {
    if (requestType == RequestType.MULTIPART) {
        throw methodError(
            "Multipart can only be specified on HTTP methods with request body (e.g.,
@POST).");
    }
    if (requestType == RequestType.FORM_URL_ENCODED) {
        throw methodError("FormUrlEncoded can only be specified on HTTP methods
with request body "
            + "(e.g., @POST).");
    }
}

```

```

    }
}
}

```

这里的 `methodAnnotation` 是 `@retrofit.http.GET(value=/myserviceD/getUser)`; `annotationType` 是 `interface retrofit.http.GET`; 之后为 `RestMethod methodInfo` 赋值, 其值实际上就是上边的 `methodAnnotation`; 然后获取到的 `path` 是 `"/myserviceD/getUser"`。之后对 `path` 进行解析, 主要是处理 `path` 上的参数, 这里不带参数, 笔者也推荐不要直接在 `path` 上使用参数, 并且该方法还为 `RestMethodInfo` 中的三个属性赋了值, 其中 `requestUrl` 被赋值为 `/myserviceD/getUser`。最后为 `RestMethodInfo` 中的两个属性 `requestMethod` 和 `requestHasBody` 赋值, 前者为 `"GET"`, 后者为 `false`。

再来看解析入参注解的源码:

```

private void parseParameters() {
    Class<?>[] methodParameterTypes = method.getParameterTypes();

    Annotation[][] methodParameterAnnotationArrays =
method.getParameterAnnotations();
    int count = methodParameterAnnotationArrays.length;
    if (!isSynchronous && !isObservable) {
        count -= 1; // Callback is last argument when not a synchronous method.
    }

    Annotation[] requestParamAnnotations = new Annotation[count];

    ...

    for (int i = 0; i < count; i++) {
        Class<?> methodParameterType = methodParameterTypes[i];
        Annotation[] methodParameterAnnotations =
methodParameterAnnotationArrays[i];
        if (methodParameterAnnotations != null) {
            for (Annotation methodParameterAnnotation : methodParameterAnnotations) {
                Class<? extends Annotation> methodAnnotationType =
                    methodParameterAnnotation.annotationType();

                if (methodAnnotationType == Path.class) {
                    String name = ((Path) methodParameterAnnotation).value();

```

```

        validatePathName(i, name);
    }
    ...
    else if (methodAnnotationType == Query.class) {
        // Nothing to do.
    }
    ...
    else {
        continue;
    }

    if (requestParamAnnotations[i] != null) {
        throw parameterError(i,
            "Multiple Retrofit annotations found, only one allowed: @%s, @%s.",
            requestParamAnnotations[i].annotationType().getSimpleName(),
            methodAnnotationType.getSimpleName());
    }
    requestParamAnnotations[i] = methodParameterAnnotation;
}
}

if (requestParamAnnotations[i] == null) {
    throw parameterError(i, "No Retrofit annotation found.");
}
}
...

this.requestParamAnnotations = requestParamAnnotations;
}

```

这里只有一个查询注解`@retrofit.http.Query(encodeName=false, encodeValue=true, value=username)`，并且将该值作为 `RestMethodInfo` 的 `requestParamAnnotations` 属性（是一个 `Annotation[]`）中的一个元素。这里列出了几乎所有的用在入参上的注解，为了节省篇幅，就省略了。

9.6.11 构建请求参数 retrofit.client.Request

首先是构建 RequestBuilder, 实际上就是将众多的信息进行汇总, 包括基本 url、converter 及 RestMethodInfo 中的各个属性。之后使用 requestBuilder.setArguments(args) 拼接输入参数。例如, getUser 方法中拼接好的参数就是 “?username=%E5%B0%8F%E8%B5%B5”, 其中值由于是中文, Retrofit 进行了 URLEncoder.encode(value, "UTF-8") 处理。之后使用 RequestBuilder 创建了 Request 对象。源码如下:

```
Request build() throws UnsupportedOperationException {
    if (multipartBody != null && multipartBody.getPartCount() == 0) {
        throw new IllegalStateException("Multipart requests must contain at least one part.");
    }

    String apiUrl = this.apiUrl;
    StringBuilder url = new StringBuilder(apiUrl);
    if (apiUrl.endsWith("/")) {
        url.deleteCharAt(url.length() - 1);
    }

    url.append(relativeUrl);

    StringBuilder queryParams = this.queryParams;
    if (queryParams != null) {
        url.append(queryParams);
    }

    TypedOutput body = this.body;
    List<Header> headers = this.headers;
    if (contentTypeHeader != null) {
        if (body != null) {
            body = new MimeOverridingTypedOutput(body, contentTypeHeader);
        } else {
            Header header = new Header("Content-Type", contentTypeHeader);
            if (headers == null) {
                headers = Collections.singletonList(header);
            } else {
                headers.add(header);
            }
        }
    }
}
```



```

    }
}

return new Request(requestMethod, url.toString(), headers, body);
}

```

这里拼接好的请求的 url 为 “http://localhost:8080/myserviceD/getUser?username=%E5%B0%8F%E8%B5%B5”，requestMethod 为 “GET”，headers 和 body 还是 null。最后，使用这 4 个值创建 Request 实例。

```

private final String method;
private final String url;
private final List<Header> headers;
private final TypedOutput body;

public Request(String method, String url, List<Header> headers, TypedOutput
body) {
    if (method == null) {
        throw new NullPointerException("Method must not be null.");
    }
    if (url == null) {
        throw new NullPointerException("URL must not be null.");
    }
    this.method = method;
    this.url = url;

    if (headers == null) {
        this.headers = Collections.emptyList();
    } else {
        this.headers = Collections.unmodifiableList(new
ArrayList<Header>(headers));
    }

    this.body = body;
}

```

9.6.12 利用 clientProvider 进行真正的调用

来看一下真正的调用源码：

```

private final OkHttpClient client;

```



```

@Override public Response execute(Request request) throws IOException {
    return parseResponse(client.newCall(createRequest(request)).execute());
}

static com.squareup.okhttp.Request createRequest(Request request) {
    com.squareup.okhttp.Request.Builder builder = new com.squareup.okhttp.
Request.Builder()
        .url(request.getUrl())
        .method(request.getMethod(), createRequestBody(request.getBody()));

    List<Header> headers = request.getHeaders();
    for (int i = 0, size = headers.size(); i < size; i++) {
        Header header = headers.get(i);
        String value = header.getValue();
        if (value == null) value = "";
        builder.addHeader(header.getName(), value);
    }

    return builder.build();
}

static Response parseResponse(com.squareup.okhttp.Response response) {
    return new Response(response.request().urlString(), response.code(),
response.message(),
        createHeaders(response.headers()),
createResponseBody(response.body()));
}

```

其实就是调用 `OkHttpClient` 进行方法调用。首先通过 `createRequest` 方法将 `retrofit.client.Request` 转换为 `com.squareup.okhttp.Request`。之后使用 `okHttpClient` 进行调用。最后，将返回的 `com.squareup.okhttp.Response` 转换为 `retrofit.client.Response`。我们实现自己的 `Client` 时可以参考该 `OkHttpClient`。这是第二个模仿点!!!

9.6.13 处理响应

最后，记录方法执行的时间，返回 `http` 响应码。如果 `http` 响应码在 `[200,300)` 之间，则认为执行成功，否则，执行失败。如果失败，直接抛出异常；如果成功，则先判断之前在接口 `API` 方法上定义的返回值是否是 `Response`，如果是，做简单处理后直接返回，不需要经过转换器；如果不是，则对返回值做反序列化操作。

第 10 章

微服务降级容错

10.1 初识 Hystrix

Hystrix 是 Netflix 开源技术栈中的又一员猛将,其主要用于在微服务中实现容错和降级。

10.1.1 为什么要使用 Hystrix

来看三个场景。

场景一：如图 10-1 所示，其中圆圈代表用户，假设有两台服务器 `server1` 和 `server2`，在 `server1` 上注册了两个服务 `service1` 和 `service2`，在 `server2` 上注册了一个服务 `service3`。假设 `service3` 服务响应缓慢，`service1` 调用 `service3` 时，一直在等待响应，那么在高并发的情况下，`server1` 处很快就会达到并发处理请求的阈值而宕机，这时候，不只是 `service1` 不再可用，`server1` 上的 `service2` 也不可用了。

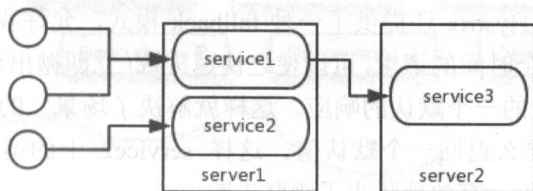


图10-1 同台机器服务级联失败

场景二：如图 10-2 所示，假设有三台服务器 server1、server2 和 server3，三台服务器上分别注册了一个服务，假设为 service1、service2 和 service3。用户发起一个请求来调用 service1，那么 service1 需要调用 service2，service2 需要调用 service3 才能处理完这个请求。假设 service3 服务响应缓慢，service2 调用 service3 时，一直在等待响应，那么在高并发的情况下，server2 处很快就会达到并发处理请求的阈值而宕机，这时候，由于 service2 响应缓慢，service1 上积压的请求也会急剧增加，导致 server1 处也可能达到请求并发处理的阈值而宕机，这样就造成了级联失败，严重的话，可能会造成雪崩。

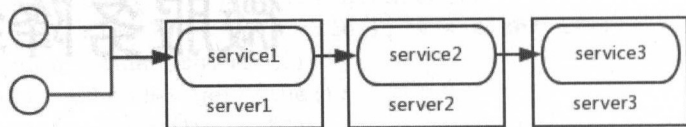


图10-2 不同机器服务级联失败

场景三：假设场景一中的 service3 崩了，那么 service3 服务恢复到正常状态可能会花比较长的一段时间，也可能需要人工来恢复。

这几个场景都是很危险的，尤其是场景二，服务雪崩绝对是企业的大灾难。为了防止出现这些问题，Netflix 又推出了一员大将：Hystrix!!! 下面我们就来看看 Hystrix 是怎样解决这些问题的。

10.1.2 Hystrix 工作原理

首先来看一下 Hystrix 官网提供的一张图。

如图 10-3 所示，Hystrix 提供了一种特殊的线程池模型，它为每一个服务提供一个线程池，其中，线程池的数量是可以配置的。这种模型其实是舱壁模式的一种实现，它将每个依赖服务隔离开来，隔离后，延时的被调用服务只会耗尽自己的线程池，之后进入失败状态；如果不隔离，可能会耗尽整个 Tomcat 的线程池，导致整个服务器挂掉。这样就解决了场景一的问题，service1 的线程池耗尽后，server1 的线程池不会耗尽，service2 依旧可用。

除了线程池模型，Hystrix 还提供了一种 fallback 模式，允许我们控制被调用服务的响应延时时间，对超出这个时间的请求，可以使之快速失败（立即抛出运行时异常结束流程），其也可以返回我们指定的一个默认的响应。这样就解决了场景二的问题，service3 响应慢，要么立即抛出异常，要么返回一个默认值，这样 service2 上的请求数量不会积压，从而 service1 上也就不会积压，有效地防止了级联失败。

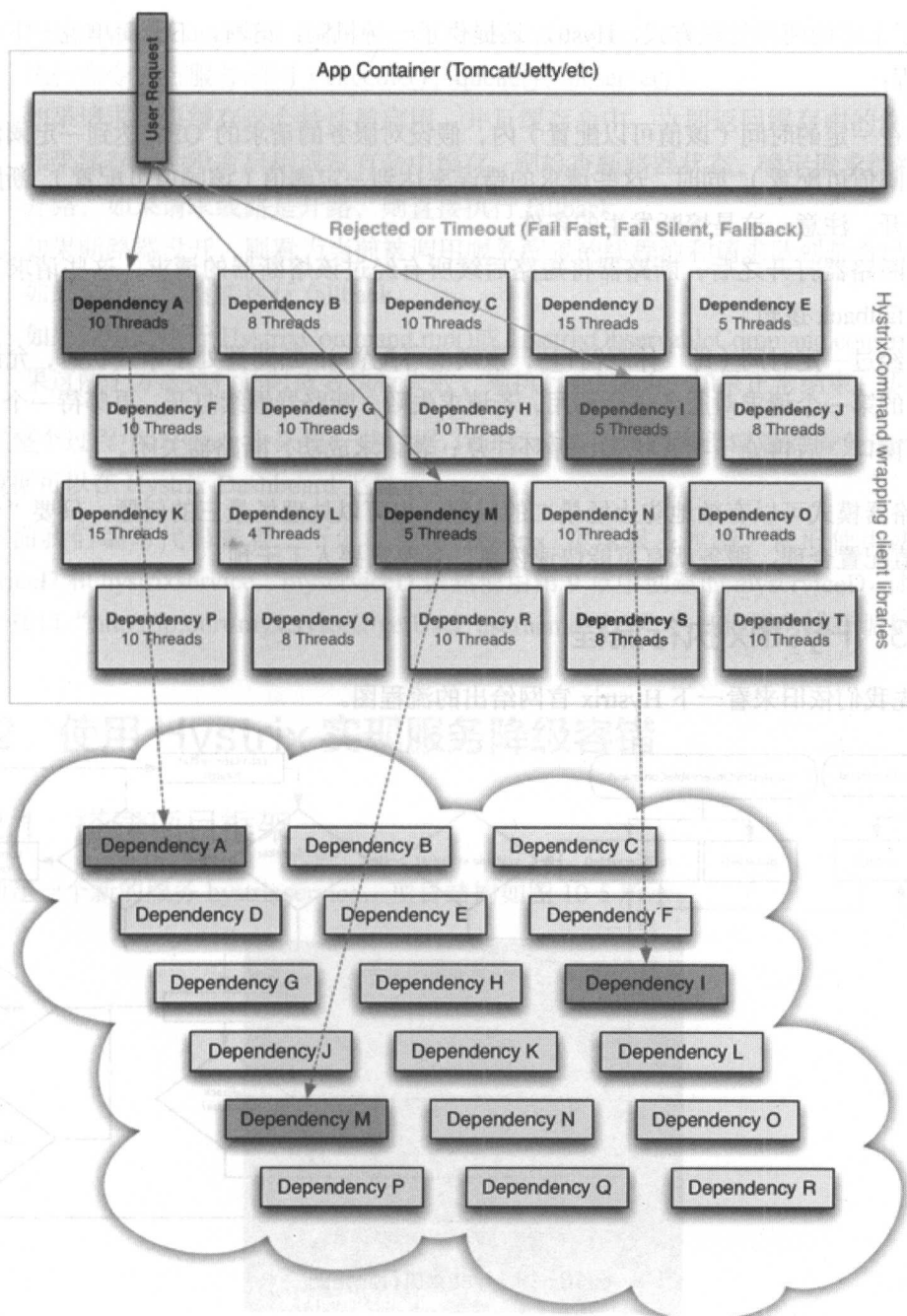


图10-3 Hystrix线程模型

除了上述的两种处理方式，Hystrix 还提供了一种机制：熔断。下面简单说一下整个熔断的过程：

1. 在一定的时间内（该值可以配置），假设对服务的请求的 QPS 达到一定阈值（该阈值可配置），同时，这些请求的错误率达到一定阈值（该阈值可配置），断路器打开。注意，这是熔断发生的条件。
2. 断路器打开之后，断路器将短路后续所有经过该熔断器的请求，这些请求直接走 fallback 逻辑。
3. 经过一定时间（即“休眠窗口”，该阈值可配置），断路器处于半开状态，允许后续的第一个请求对服务进行调用，若请求失败，断路器继续打开，再等待一个“休眠窗口”后再处于半开状态，循环往复；若请求成功，断路器关闭。

断路器模式可以有效地防止场景二的问题；也可以处理场景三的问题，只要“休眠窗口”的值配置合理，服务就有可能快速恢复，且不需要人工干预。

10.1.3 Hystrix 执行流程

首先我们依旧来看一下 Hystrix 官网给出的流程图。

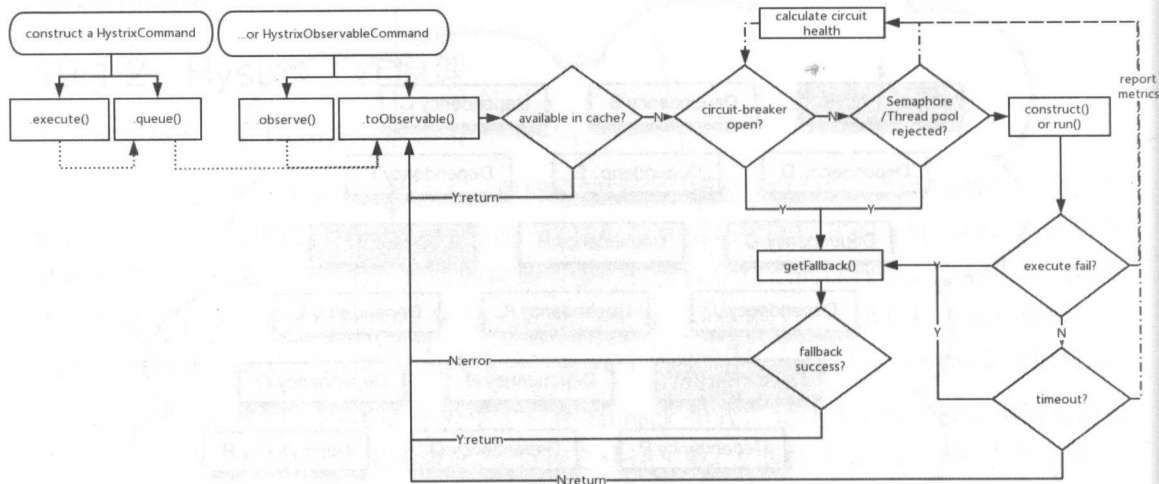


图10-4 Hystrix执行流程图

如图 10-4 所示，Hystrix 的执行流程如下：

1. 构建 `HystrixCommand` 或者 `HystrixObservableCommand` 对象。
2. 执行命令进行服务调用 (`execute()`、`queue()`、`observe()`)。
3. 如果请求结果缓存这个特性被启用, 并且缓存命中, 立即返回缓存中的值。
4. 如果缓存特性没有启用或没有命中缓存, 则检查断路器状态, 确定请求线路是否是开路, 如果请求线路是开路, 则直接执行 `fallback`。
5. 如果断路器没开, 则看为当前被调用服务配置的线程池和请求队列是否已经满了, 如果满了, 则直接执行 `fallback`。
6. 如果没满, 执行 `HystrixCommand.run()`或 `HystrixObservableCommand.construct()`; 如果这两个方法执行超时或者执行失败, 则执行 `fallback`; 如果正常结束, 返回响应。

在整个过程中, `Hystrix` 会采集一些数据, 以进行后续的熔断操作或者对服务进行监控。这些数据可以在 `Hystrix-Dashboard` 上显示。

下面我们编写代码来看一下怎样使用 `Hystrix` 实现服务容错。依旧使用两个服务 `myserviceD` 和 `hystrixservice`, `myserviceD` 依然使用第 9 章中创建的 `myserviceD`, 其仅仅提供一个接口 “`http://ip:port/myserviceD/getUser?username=xxx`” 供其他服务进行调用。

10.2 使用 `Hystrix` 实现服务降级容错

10.2.1 搭建项目框架

创建一个新的服务 `hystrixservice`, 项目结构如图 10-5 所示。

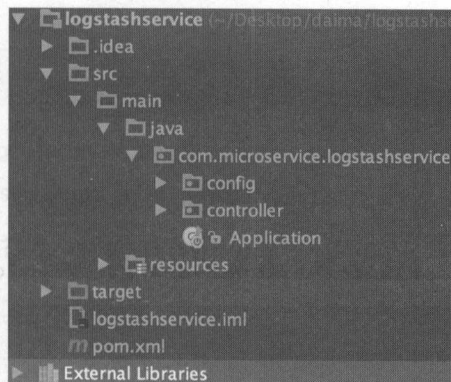


图10-5 `hystrixservice`服务项目结构

其中，pom.xml 文件的内容如下：

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/maven-v4_0_0.xsd">
  <modelVersion>4.0.0</modelVersion>

  <groupId>com.microservice</groupId>
  <artifactId>hystrixservice</artifactId>
  <version>1.0-SNAPSHOT</version>

  <properties>
    <java.version>1.8</java.version>
  </properties>
  <parent>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-parent</artifactId>
    <version>1.4.3.RELEASE</version>
  </parent>

  <dependencies>
    <dependency>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-starter-web</artifactId>
    </dependency>
    <dependency>
      <groupId>io.springfox</groupId>
      <artifactId>springfox-swagger2</artifactId>
      <version>2.2.2</version>
    </dependency>
    <dependency>
      <groupId>io.springfox</groupId>
      <artifactId>springfox-swagger-ui</artifactId>
      <version>2.2.2</version>
    </dependency>
    <dependency>
      <groupId>org.projectlombok</groupId>
      <artifactId>lombok</artifactId>
```

```

        <version>1.16.8</version>
    </dependency>
    <dependency>
        <groupId>com.alibaba</groupId>
        <artifactId>fastjson</artifactId>
        <version>1.1.15</version>
    </dependency>
    <!-- async-http-client -->
    <dependency>
        <groupId>com.ning</groupId>
        <artifactId>async-http-client</artifactId>
        <version>1.9.31</version>
    </dependency>
    <!-- hystrix -->
    <dependency>
        <groupId>com.netflix.hystrix</groupId>
        <artifactId>hystrix-core</artifactId>
        <version>1.5.9</version>
    </dependency>
</dependencies>

<build>
    <plugins>
        <plugin>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-maven-plugin</artifactId>
        </plugin>
    </plugins>
</build>
</project>

```

除了熟悉的 `spring-boot-starter-web`、`Swagger`、`Lombok`、`fastjson` 及 `async-http-client` 的依赖外，还引入了以下 `hystrix-core` 依赖，以提供 `Hystrix` 的核心接口。

引入依赖之后，创建主类，代码如下：

```

package com.microservice.hystrixservice;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;

```

```
import springfox.documentation.swagger2.annotations.EnableSwagger2;

@SpringBootApplication
@EnableSwagger2
public class Application {
    public static void main(String[] args) {
        SpringApplication sa = new SpringApplication(Application.class);
        sa.run(args);
    }
}
```

依旧是熟悉的启动主类。

项目框架搭建完成之后，编写代码实现服务降级容错功能。

10.2.2 创建 AsyncHttpClient 调用实体类

服务通信框架依然使用 AsyncHttpClient，需要创建一个 AsyncHttpClient 单例 Bean，代码如下：

```
package com.microservice.hystrixservice.config;

import com.ning.http.client.AsyncHttpClient;
import com.ning.http.client.AsyncHttpClientConfig;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;

@Configuration
public class HystrixServiceConfig {
    @Bean
    public AsyncHttpClient asyncHttpClient() {
        AsyncHttpClientConfig.Builder builder = new
        AsyncHttpClientConfig.Builder()
            .setRequestTimeout(50000)
            .setConnectTimeout(20000)
            .setReadTimeout(30000);
        return new AsyncHttpClient(builder.build());
    }
}
```


10.2.3 服务通信框架集成服务降级容错功能

为了实现服务降级容错功能，需要使用 `HystrixCommand` 实现类来包裹服务通信框架的调用行为，具体代码如下：

```
package com.microservice.hystrixservice.config;

import com.netflix.hystrix.HystrixCommand;
import com.ning.http.client.AsyncHttpClient;
import com.ning.http.client.RequestBuilder;
import com.ning.http.client.Response;

import java.util.concurrent.ExecutionException;
import java.util.concurrent.Future;

public class MyHystrixCommand extends HystrixCommand<Response> {
    private String url;
    private AsyncHttpClient asyncHttpClient;

    public MyHystrixCommand(Setter setter, String url, AsyncHttpClient
asyncHttpClient) {
        super(setter);
        this.url = url;
        this.asyncHttpClient = asyncHttpClient;
    }

    @Override
    public Response run() throws Exception {
        Response response = null;
        com.ning.http.client.Request request = new
RequestBuilder().setUrl(url).build();
        Future<Response> responseFuture =
asyncHttpClient.executeRequest(request);
        try {
            response = responseFuture.get();
        } catch (InterruptedException | ExecutionException e) {
            e.printStackTrace();
        }
        return response;
    }
}
```



```

@Override
public Response getFallback() {
    throw new RuntimeException("active hystrix getFallback");//实现快速失败
}
}

```

该类继承了 `com.netflix.hystrix.HystrixCommand`，且指定了泛型参数为 `com.ning.http.client.Response`，该值也是 `run()`方法和 `getFallBack()`方法的返回值。`run()`方法是正常调用的方法，在该方法内，我们使用 `asyncHttpClient` 执行对 `myserviceD` 的调用；`getFallBack()`方法是 `fallback` 模式的方法，就是 `run()`方法执行失败或者超时时调用的方法，其可以返回一个自定义的 `com.ning.http.client.Response` 实例，也可以抛出异常，让请求快速失败。

关于该类还有一个值得注意的事情，就是必须要有构造器，而且在构造器中必须调用 `super(setter)`方法，在该方法中，会为当前的 `MyHystrixCommand` 实例初始化一系列的配置信息，例如 `commandKey` 等。

下面，编写一个集成服务降级容错功能的服务通信类，代码如下：

```

package com.microservice.hystrixservice.config;

import com.netflix.hystrix.*;
import com.ning.http.client.AsyncHttpClient;
import com.ning.http.client.Response;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Component;

@Component
public class HystrixUtil {
    @Autowired
    private AsyncHttpClient asyncHttpClient;

    public Response execute (String serviceName, String methodName, String url) {
        HystrixCommand.Setter setter = HystrixCommand.Setter.withGroupKey
            (HystrixCommandGroupKey.Factory.asKey(serviceName));
        setter.andCommandKey(HystrixCommandKey.Factory.asKey(serviceName+"."+methodName));
        setter.andCommandPropertiesDefaults(HystrixCommandProperties.Setter().withExecutionTimeoutInMilliseconds(3000));
        setter.andThreadPoolPropertiesDefaults(HystrixThreadPoolProperties.

```

```

Setter().withCoreSize(20));

        return new MyHystrixCommand(setter,url,asyncHttpClient).execute();
    }
}

```

我们在这里定义了一个 `execute` 方法，在该方法中，先创建了一个 `com.netflix.hystrix.HystrixCommand.Setter` 实例，之后为该实例设置了 `commandGroupKey`、`commandKey`、超时时间（这里设为 3 s），以及为 `hystrixservice` 服务调用 `myserviceD` 创建的线程池的核心线程数 `coreSize`。

`commandGroupKey` 通常被设为 `servicename`；而如果没有自己指定 `threadPoolKey` 的值，则其通常会取 `commandGroupKey` 的值；`commandKey` 通常会以方法名命名，为了防止多个服务的方法重名，通常会使用 `servicename` 前缀。

该 `setter` 最后会用来在创建 `MyHystrixCommand` 实例时，为其设置属性值，具体看一下 `com.microservice.hystrixservice.config.MyHystrixCommand` 的构造器的 `super(setter)` 的源码，这里不再赘述。

最后，我们来验证降级容错功能。

10.2.4 验证服务降级容错功能

首先，创建一个模型类来接收 `myserviceD` 被调用接口的返回值。代码如下：

```

package com.microservice.hystrixservice.model;

import lombok.Getter;
import lombok.Setter;

@Setter
@Getter
public class User {
    private String username;
    private int    age;
}

```

然后编写一个 `controller` 进行调用。

```

package com.microservice.hystrixservice.controller;

```

```

import com.alibaba.fastjson.JSON;
import com.microservice.hystrixservice.config.HystrixUtil;
import com.microservice.hystrixservice.model.User;
import com.ning.http.client.Response;
import io.swagger.annotations.Api;
import io.swagger.annotations.ApiOperation;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestMethod;
import org.springframework.web.bind.annotation.RequestParam;
import org.springframework.web.bind.annotation.RestController;
import java.io.IOException;

@Api("服务容错相关 api")
@RestController
@RequestMapping("/hystrix")
public class TestController {
    @Autowired
    private HystrixUtil hystrixUtil;

    @ApiOperation("根据用户名获取用户信息")
    @RequestMapping(value = "/getUserByName", method = RequestMethod.GET)
    public User getUserByName(@RequestParam("username") String username) {
        User user = null;

        String url = "http://localhost:8080/myserviceD/getUser?username=" +
username;
        Response response = hystrixUtil.execute("myserviceD", "getUser", url);
        if (response != null) {
            try {
                user = JSON.parseObject(response.getResponseBody(), User.class);
            } catch (IOException e) {
                e.printStackTrace();
            }
        }
        return user;
    }
}

```

代码编写完成后，部署服务进行测试。启动 myserviceD 服务，然后启动 hystrixservice 服务，进入 Swagger 进行测试!!! 这时候，由于 Hystrix 的超时时间是 3 s，而 myserviceD 相应也比较快，所以会返回正常的结果。我们对 myserviceD 的 getUser 方法进行一点修改，如下：

```
@ApiOperation("根据用户名获取用户信息")
@RequestMapping(value = "/getUser", method = RequestMethod.GET)
public User getUser(@RequestParam("username") String username) {
    try {
        TimeUnit.MILLISECONDS.sleep(5000); //用于测试超时
    } catch (InterruptedException e) {
        e.printStackTrace();
    }

    User user = null;
    if (username.equals("小娜")) {
        user = new User(username, 18);
    } else {
        user = new User("小刚", 20);
    }
    return user;
}
```

该方法在执行过程中首先会睡 5 s，之后才会执行后续逻辑。这样的话，已经超出了 Hystrix 的超时时间 (3 s)，所以会执行 fallback 逻辑，直接抛出异常。在 response body 中会返回如下信息：

```
{
  "timestamp": 1484885473541,
  "status": 500,
  "error": "Internal Server Error",
  "exception": "com.netflix.hystrix.exception.HystrixRuntimeException",
  "message": "myserviceD:getUser timed-out and fallback failed.",
  "path": "/hystrix/getUserByName"
}
```

其中 message 是我们自己指定的报错信息。

最后，笔者要指出一个坑 (是个隐秘的坑)，就是 Hystrix 的超时时间和 AsyncHttpClient

请求超时时间的设置，记住一点，以二者的最小值为准。在本例中，笔者为 Hystrix 设置的超时时间是 3s，而 AsyncHttpClient 的 requestTimeout 为 50 s。所以整个请求的超时时间是 3 s，而如果为 Hystrix 设置的超时时间是 30 s，AsyncHttpClient 的 requestTimeout 为 20 s，则整个请求的超时时间是 20s。由于 AsyncHttpClient 的调用是在 run()方法中进行的，当整个请求超过 20s 时，会出现 AsyncHttpClient 调用超时错误，也就是 run()执行错误，所以会马上执行 getFallback()方法。

10.3 搭建 Hystrix 监控系统

如图 10-4 所示，Hystrix 在执行过程中，会监控记录一些有用的数据，至于记录了哪些数据，Hystrix-Dashboard 会向我们展示。使用这些数据我们可以更好地来设置 Hystrix 的众多属性值，使系统达到最优的状态。Hystrix 将最近 10 s 内（当然，这个值也是可以配置的）的数据存储在内存中的一个滚动桶模型中（该模型有 10 个桶，每个桶记录 1s 的数据），这里引用 Hystrix 官网的一张图片，如图 10-6 所示。

Success	23	47	26	48	38	42	59	46	39	12
Failure	5	8	4	9	4	6	11	5	3	1
Timeout	2	1	0	4	2	7	5	2	5	0
Rejection	0	0	0	0	0	0	1	0	0	0

10 1-second "buckets"

图10-6 Hystrix-Metrics滚动桶模型

从这个桶中我们只能拿到最近 10 s 的数据，再往前的数据只能通过分析 HystrixRequestLog 来获得了。

10.3.1 使用 Hystrix-Metrics-Event-Stream 发布监控信息

想使用 Hystrix-Metrics 将统计信息发布出来，只需要在 hystrixservice 服务中进行如下两步操作即可。

第一步，在 pom.xml 中引入 hystrix-metrics-event-stream 依赖，如下：

```
<!-- hystrix-metrics-event-stream -->
<dependency>
  <groupId>com.netflix.hystrix</groupId>
  <artifactId>hystrix-metrics-event-stream</artifactId>
```



```
<version>1.5.9</version>
</dependency>
```

第二步，创建 `HystrixMetricsStreamServlet` 实例并注册，代码如下：

```
@Bean
public HystrixMetricsStreamServlet hystrixMetricsStreamServlet() {
    return new HystrixMetricsStreamServlet();
}

@Bean
public ServletRegistrationBean registration(HystrixMetricsStreamServlet
servlet) {
    ServletRegistrationBean registrationBean = new
ServletRegistrationBean();
    registrationBean.setServlet(servlet);
    registrationBean.addUrlMappings("/hystrix.stream");
    return registrationBean;
}
```

在 `HystrixServiceConfig` 中添加两个 Bean：一个是 `HystrixMetricsStreamServlet`；另一个是用来注册 Servlet 的 `ServletRegistrationBean`，并且指定了注册的 Servlet 拦截的 `urlMappings`。

至此代码部分就完成了，只要 `Hystrix-Dashboard` 连接一直连着，`Hystrix-Metrics-Event-Stream` 就会不断地向客户端以 `text/event-stream` 的形式推送计数结果（metrics）。

10.3.2 使用 Hystrix-Dashboard 展示监控信息

使用 `Hystrix-Dashboard` 有三种姿势：

- 下载 `Hystrix-Dashboard-###.war` 包，并放到 Tomcat 中，之后以 Web 项目启动。
- 下载 `Hystrix-Dashboard` 的 `fatjar`，之后直接以 `jar` 包方式启动。
- 下载 `Hystrix-Dashboard` 的 Docker 镜像，以 Docker 镜像启动。

其中，第二种最简单，这里使用第二种方式。

第一步，在本机下载 `standalone-hystrix-dashboard-1.5.3-all.jar`，下载地址为：<https://github.com/kennedyoliveira/standalone-hystrix-dashboard>。

第二步，将下载好的 `jar` 包复制到 10.211.55.13（笔者的一台 centos7 的虚拟机）。

```
scp standalone-hystrix-dashboard-1.5.3-all.jar root@10.211.55.13:/opt/
```

第三步，启动 Hystrix-Dashboard。

```
nohup java -jar -DserverPort=7979 -DbindAddress=10.211.55.13
standalone-hystrix-dashboard-1.5.3-all.jar &
```

其中，serverPort 的默认值是 7979，bindAddress 的默认值是 localhost。

第四步，在浏览器中输入：<http://10.211.55.13:7979/hystrix-dashboard/>，出现小熊页面就表明正确运行了。

第五步，在小熊页面按照图 10-7 所示的流程进行操作。

1

Hystrix Dashboard

<http://localhost:8081/hystrix.stream>

Cluster via Turbine (default cluster): <http://turbine-hostname:port/turbine.stream>
 Cluster via Turbine (custom cluster): [http://turbine-hostname:port/turbine.stream?cluster=\[clusterName\]](http://turbine-hostname:port/turbine.stream?cluster=[clusterName])
 Single Hystrix App: <http://hystrix-app:port/hystrix.stream>

Delay: 2000 ms Title: Example Hystrix App

Authorization: Basic Zm9vOmJhbm91

2 Add Stream

3 Monitor Streams

图10-7 创建Hystrix-Stream

首先在输入框输入 `hystrixservice` 的监控地址：`http://host:port/hystrix.stream`，然后单击 `Add Stream` 按钮，最后单击 `Monitor Streams` 按钮即可。这里注意，如果 `host` 用 `localhost` 或 `127.0.0.1` 不行的话，需要使用真实 IP。单击 `Monitor Streams` 按钮之后，会看到如图 10-8 所示的页面。

Hystrix Stream: <http://10.10.10.10:8081/hystrix.stream>

Circuit Sort: [Error then Volume](#) | [Alphabetical](#) | [Volume](#) | [Error](#) | [Mean](#) | [Median](#) | [90](#) | [99](#) | [99.5](#) Success | [Short-Circuited](#) | [Bad Request](#) | [Timeout](#)

myserviceD:getUser

0	0	0.0 %
0	0	
0	0	

Host: **0.0/s**

Cluster: **0.0/s**

Circuit Closed

Hosts	1	90th	9ms
Median	6ms	99th	17ms
Mean	6ms	99.5th	17ms

Thread Pools Sort: [Alphabetical](#) | [Volume](#) |

myserviceD

Host: **0.0/s**

Cluster: **0.0/s**

Active	0	Max Active	0
Queued	0	Executions	0
Pool Size	20	Queue Size	5

图10-8 Hystrix-Stream监控信息

可以看到, Hystrix 监控了每一个 `commandKey` 和 `threadPool`。这里通过 Hystrix 官方的一张图片来看一下 `Circuit` 的仪表盘中的各个值代表什么意思（其实可以根据 Hystrix-Dashboard 页面右上方的颜色来判断每一个参数代表什么意思）。

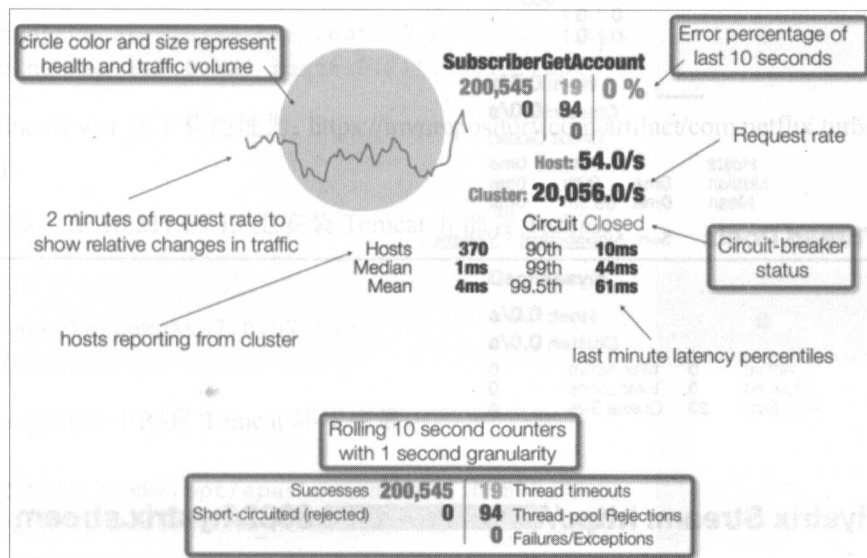


图10-9 Hystrix-Dashboard仪表盘

读者可以使用 Swagger 进行测试, 看看 Hystrix-Dashboard 上的数据变化。该页面是动态刷新的, 但是时间久了也会卡死, 如果页面没有刷新, 手动刷新一下。

10.3.3 使用 Turbine 聚合监控信息

Turbine 是 Netflix 开源技术栈中的又一力作, 其主要用于聚合数据。在 Hystrix 中, 其主要用于将相同的 `commandKey`、`Threadpool` 等监控数据进行聚合。

具体可以看一下这个场景: 假设现在有两个服务 `hystrixservice` 和 `hystrixservice2` (`hystrixservice2` 的代码从 `hystrixservice` 复制而来) 都要访问 `myserviceD` 服务的 `getUser` 方法。那么在 Hystrix-Dashboard 中 `myserviceD:getUser` 这个 `commandKey` 以及 `myserviceD` 这个 `threadPool` 会分开在两个流中展示, 如图 10-10 所示。

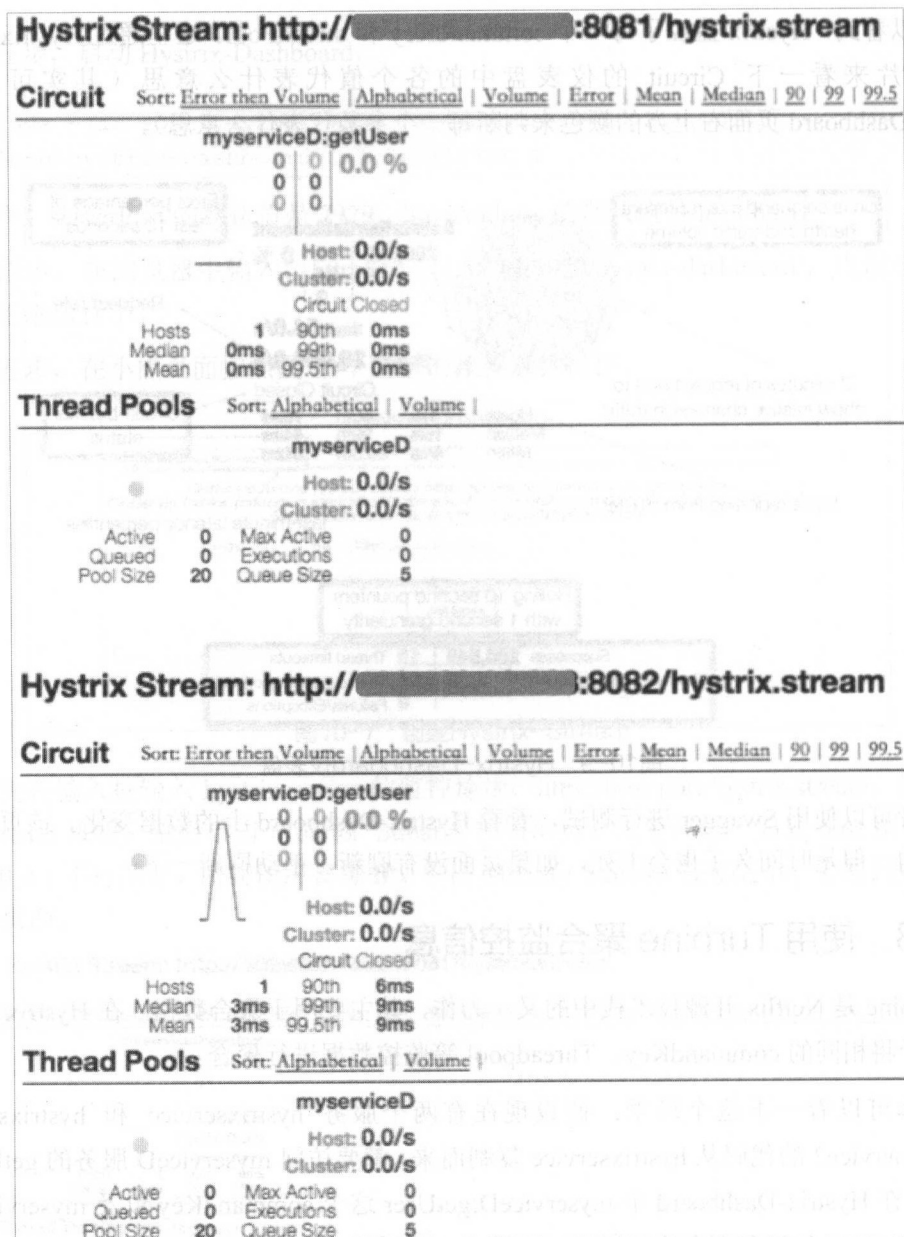


图10-10 未使用Turbine前的多服务监控信息图

实际上我们只需要统计 myserviceD 服务的 getUser 方法的处理能力，与谁调用无关，所以希望让相同的 commandKey 或者相同的 threadPool 展示在一个仪表盘中，这样才可以

看出这个方法或者这个服务的整体响应能力。这个时候就需要使用 Turbine 将这些数据汇聚起来。使用 Turbine 需要执行如下步骤：

第一步，从本地下载 Tomcat 和 Turbine 的 war 包，之后上传到 10.211.55.13。

```
scp apache-tomcat-7.0.57.tar root@10.211.55.13:/opt/
scp turbine-web-1.0.0.war root@10.211.55.13:/opt/
```

Turbine 的 war 包下载地址为：<https://mvnrepository.com/artifact/com.netflix.turbine/turbine-web/1.0.0>。

第二步，在 10.211.55.13 上安装 Tomcat 并部署 Turbine。

```
cd /opt/
tar -xf apache-tomcat-7.0.57.tar
vi /etc/profile
```

在/etc/profile 中配置 Tomcat 环境变量。

```
export TOMCAT_HOME=/opt/apache-tomcat-7.0.57
export PATH=$PATH:$TOMCAT_HOME/bin
```

配置完成之后，使配置文件生效，然后部署 Turbine。

```
source /etc/profile
cp turbine-web-1.0.0.war apache-tomcat-7.0.57/webapps/
vi /opt/apache-tomcat-7.0.57/conf/server.xml
```

在 Host 标签内添加相关内容。

```
<Context path="/" docBase="turbine-web-1.0.0.war" debug="0" privileged="true"
reloadable="true"/>
```

最后，启动 Tomcat。

```
cd /opt/apache-tomcat-7.0.57/bin
nohup ./startup.sh &
```

此时，turbine-web-1.0.0.war 就会完成自动解压。我们在浏览器中输入 <http://10.211.55.13:8080/turbine-web-1.0.0/turbine.stream>，如果显示“HTTP Status 404 - Cluster not found”，则表示 turbine-web-1.0.0.war 部署成功!!! 当然也可以先手动解压 turbine-web-1.0.0.war，执行

完下边的第三步后再启动 Tomcat。

第三步，配置 Turbine。

```
vi /opt/apache-tomcat-7.0.57/webapps/turbine-web-1.0.0/WEB-INF/classes/  
config.properties
```

配置以下 4 项：

```
InstanceDiscovery.impl=com.netflix.turbine.discovery.ConfigPropertyBasedDisc  
overy  
turbine.aggregator.clusterConfig=default  
turbine.instanceUrlSuffix=:8081/hystrix.stream  
turbine.ConfigPropertyBasedDiscovery.default.instances=xx.xx.xx.xxx,10.211.5  
5.13
```

说明：

- turbine.aggregator.clusterConfig 是 Turbine 汇聚数据的单位，这里只配置了一个 default，那么 Turbine 会对所有的项目进行汇聚。
- turbine.ConfigPropertyBasedDiscovery.default.instances 与 turbine.instanceUrlSuffix 共同构建获取 Hystrix-Stream 的 url，这里是两个：http://xx.xx.xx.xxx:8081/hystrix.stream 和 http://10.211.55.13:8081/hystrix.stream。最后，Turbine 会将这两个 Hystrix-Stream 汇聚起来。其中 xx.xx.xx.xxx 和 10.211.55.13 分别部署 hystrixservice 和 hystrixservice2。

第四步，重新启动 Tomcat。

```
ps -ef | grep tomcat  
kill -9 <tomcatpid>  
cd /opt/apache-tomcat-7.0.57/bin  
nohup ./startup.sh &
```

第五步，在 Hystrix-Dashboard 上添加 Turbine-Stream，如图 10-11 所示。

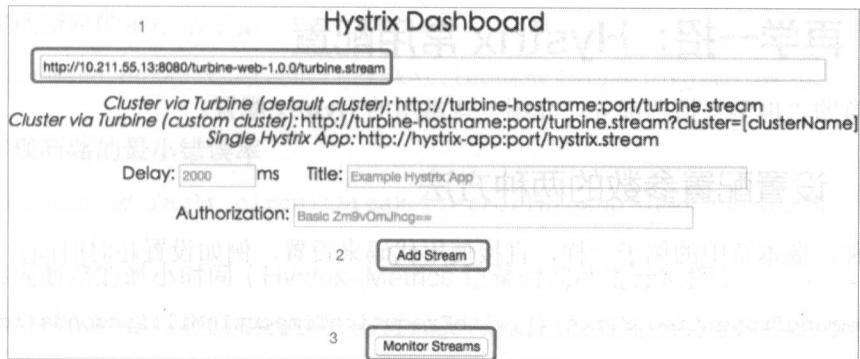


图10-11 添加Turbine-Stream

单击 Monitor Streams 按钮之后，显示结果如图 10-12 所示。

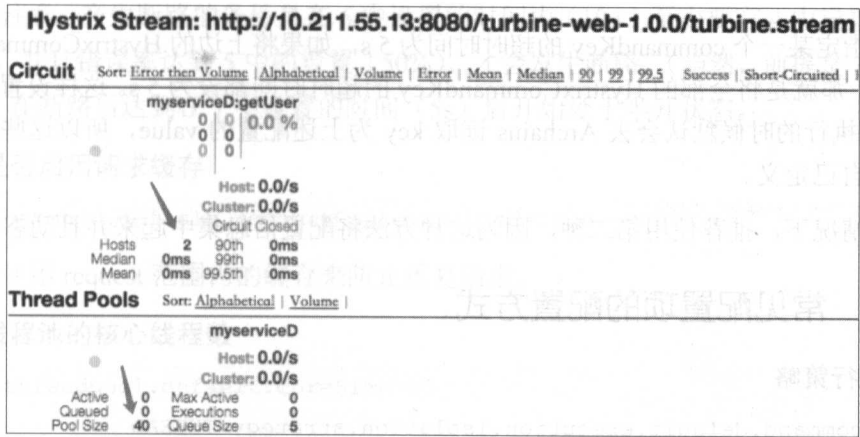


图10-12 Turbine聚合多服务Hystrix-Stream

Hosts 变为了 2，PoolSize 变为了 40，这说明 hystriservice 和 hystriservice2 两个服务汇聚成功。

至此，一套 Hystrix 监控系统就搭建成功了!!! 当然，Turbine 的聚合也是实时的。

这里笔者还遗留了一个问题，就是在实际开发中，我们不可能每添加一个服务或机器就去修改 Turbine 的 config.properties 配置文件，而是需要对 Turbine 的代码进行二次开发，通过服务发现来做这个事儿。

10.4 再学一招：Hystrix 常用配置

在本章的“再学一招”部分，介绍一些常用的 Hystrix 配置。

10.4.1 设置配置参数的两种方法

第一种，像本章中的例子一样，直接使用代码来设置，例如设置超时时间：

```
HystrixCommandProperties.Setter().withExecutionTimeoutInMilliseconds(int value)
```

第二种，当项目使用 Archaius 进行服务配置的时候，可以直接在配置文件中设置，

```
hystrix.command.HystrixCommandKey.execution.isolation.thread.timeoutInMilliseconds=5000
```

这是指定某一个 commandKey 的超时时间为 5 s，如果将上边的 HystrixCommandKey 换成 default，那就是将全部的 HystrixCommandKey 的超时时间都设为 5 s。这样设置了之后，Hystrix 在执行的时候默认会去 Archaius 读取 key 为上述配置的 value，所以这些 key 要写对而不能自己定义。

通常情况下，推荐使用第二种，因为这种方法将配置信息集中起来并且动态管理了。

10.4.2 常见配置项的配置方式

1. 执行策略

```
hystrix.command.default.execution.isolation.strategy=THREAD
```

可选值：THREAD、SEMAPHORE。使用线程池模型的好处在 10.1.2 节中已经讲了。坏处就是占用太多的内存，但是在绝大多数情况下，Netflix 更偏向于使用线程池来隔离依赖服务，因为其带来的额外开销是可以接受的，并且它能支持包括超时在内的所有功能。通常不需要使用信号量。

2. 超时时间

```
hystrix.command.default.execution.isolation.thread.timeoutInMilliseconds=1000
```

3. 是否启用断路器

```
hystrix.command.default.circuitBreaker.enabled=true
```

4. 实现断路的最小请求量

```
hystrix.command.default.circuitBreaker.requestVolumeThreshold=20
```

5. 实现断路的最小错误率

```
hystrix.command.default.circuitBreaker.errorThresholdPercentage=50
```

6. 实现断路的最小时间 (Hystrix-Metrics 记录时间的统计数据)

```
hystrix.command.default.metrics.rollingStats.timeInMilliseconds=10000
```

7. 断路器打开后多久开始处于半开状态

```
hystrix.command.default.circuitBreaker.sleepWindowInMilliseconds=5000
```

这里注意,产生断路的条件是在 6 中设置的时间内 (10 s) 至少要有 4 中设置的请求数 (20 个),并且错误率达到 5 中的设置 (50%),才会发生断路。(当然,前提是 3 中的设置为 true),在断路后达到在 7 中设置的时间 (5s) 后开始处于半开状态。

8. 是否启用请求缓存

```
hystrix.command.default.requestCache.enabled=true
```

通过使用 request 范围内的缓存来防止重复请求。

9. 线程池的核心线程数

```
hystrix.threadpool.default.coreSize=10
```

10. 队列保存的最多的元素个数

```
hystrix.threadpool.default.maxQueueSize=-1
```

-1: 代表使用 SynchronousQueue, 那么当 coreSize 满了之后,再来的请求就直接回绝了; 正整数: 代表使用 LinkedBlockingQueue, 并且队列最大值为指定的正整数。

11. 队列动态变化大小

```
hystrix.threadpool.default.queueSizeRejectionThreshold=5
```

注意：当在 10 中的设置为 -1 时，该项不起作用；当在 10 中设置的值为正数时，例如 100，线程池就会创建一个长度为 100 的队列，且该队列的大小不能动态变化，所以如果不考虑 11 配置项的话，当同时过来 $\text{coreSize}+100$ 个请求时，此时再过来的请求就会被回绝；如果配置了 11 配置项，假设为 5，那么超过 $\text{coresize}+5$ 个请求时，再来的请求就会被回绝。起到了一个相当于动态更改队列大小的作用。

以上就是最常用的 11 个配置项，其他使用默认值就可以了。

第 11 章 微服务日志系统

第 12 章 微服务全链路追踪系统

第 13 章 微服务持续集成与持续部署系统

第 11 章

微服务日志系统

11.1 初识 ELK

ELK 是一个技术栈，包括 Elasticsearch、Logstash 和 Kibana，ELK 是三种技术首字母缩写。ELK 主要用于日志收集、存储与查询。

11.1.1 为什么要用 ELK

ELK 主要用于处理日志。那么要日志有什么用呢？本来干干净净的代码，只要注重业务就好了，现在却要在各个关键位置加上一行用于记录日志的代码，这是为啥？因为日志很重要！通过查看日志，很多事就会变得简单，比如说，

- **定位问题：**线下也许可以通过 debug 一行一行来调试，但是线上可能不会给你远程调试的权限；当然，如果代码庞大，调试可能也是很费劲的。通常我们会通过打印一些 error 级别的日志（尤其是在 try-catch 的 catch 块中，将堆栈信息打印出来）来定位一些问题。
- **分析性能：**在调用一个方法时，我们想分析一下该方法的性能怎么办？可以这样做，在方法开始执行时记录一个开始时间，在方法结束时使用 info 级别的日志记录结束时间与之前的开始时间的差值，通过这样的记录，可以看出该方法执行时间的长短，进而判断该方法是否是性能瓶颈，决定是否需要优化该方法。

- **数据挖掘：**做数据挖掘需要大量的有价值的数 据。比如我们要制定一个广告投放策略，想为不同的用户投放不同的广告，这就需要为用户打一些标签，尤其是兴趣标签。打标签就可以通过分析日志来实现（当然，埋点也可以）。

日志很重要，但是人工分析日志可能不是一件容易的事。假设一个服务只部署在一台机器上，我们分析一份日志就好了，但是如果这个服务被部署在了 6 台机器上，那么就需要分析 6 份日志，那就不好办了。这个时候该 ELK 出场了！使用 Logstash 将日志收集在一起，使用 Elasticsearch 将日志存储起来并提供搜索接口，使用 Kibana 进行日志的查询并展示。非常简单地就解决了多机器多份日志的问题。当然，人工分析日志的难点不只这一个，比如还有日志格式不统一等问题，这些都可以通过 ELK 来解决。

11.1.2 ELK 最常用的两种架构

ELK 最常用的两种架构为最简架构和缓冲架构。

其中，最简架构如图 11-1 所示。

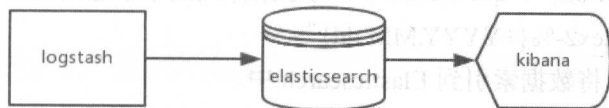


图11-1 ELK最简架构

最简架构就是使用 Logstash 收集日志，之后将日志直接存储到 Elasticsearch 中，最后用户通过 Kibana 查询日志并展示。在并发日志量比较小的情况下，使用该架构是没问题的。但是随着并发日志量的增加，由于 Logstash 将数据索引到 Elasticsearch 比较慢，如果索引失败，数据还会丢失，因此有了 ELK 的第二种架构，该架构也是企业中最常用的架构：缓冲架构。

缓冲架构如图 11-2 所示。

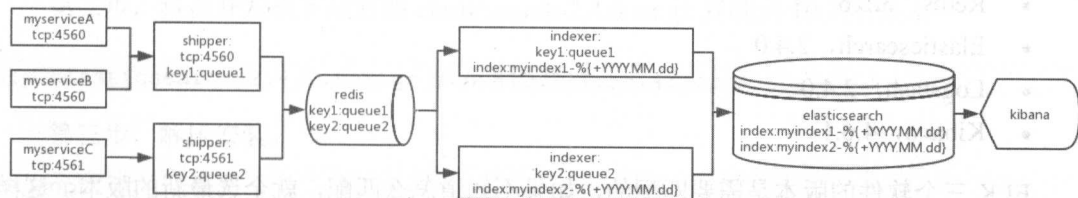


图11-2 ELK缓冲架构

很明显，缓冲架构要比最简架构复杂。实际上相较于最简架构，缓冲架构只做了两件事。第一件，将原本的 Logstash 按职能分为了 Logstash-Shipper（简称 Shipper）和 Logstash-Indexer（简称 Indexer），其中 Shipper 用于收集日志，Indexer 用于指定索引；第二件，在 Shipper 和 Indexer 之间添加了 Redis 来做缓冲，减小 Indexer 将数据索引到 Elasticsearch 的压力。

现在的流程是这样的：Shipper 进行日志收集，之后传到 Redis 中，Redis 再将数据弹出给 Indexer，Indexer 将数据索引到 Elasticsearch，最后用户使用 Kibana 查询日志。

流程依旧很简单，但是笔者在这里给出的架构图包含更多的信息，我们来看一下流程：

1. 假设有三个服务，其中 myserviceA 和 myserviceB 的日志会以 tcp 协议发到 Shipper 的 4560 端口，而 myserviceC 会发到 4561 端口。
2. 4560 的 Shipper 将数据发到 Redis 的 queue1 队列中，4561 的 Shipper 发到 queue2 队列。
3. queue1 队列中的数据弹到图中上边的 Indexer 中，并指定索引名为“myindex1-%{+YYYY.MM.dd}”，queue2 队列中的数据弹到图中下边的 Indexer 中，并指定索引名为“myindex2-%{+YYYY.MM.dd}”。
4. 两个 Indexer 将数据索引到 Elasticsearch 中。
5. 在 Kibana 中选定不同的索引来分别查看三个服务的日志。

下面我们就使用 ELK 缓冲架构搭建一套日志系统。

11.2 搭建 ELK 系统

安装环境及软件版本如下：

- 系统，centos7；ip 是 10.211.55.4
- Redis，3.2.6
- Elasticsearch，2.4.0
- Logstash，2.4.0
- Kibana，4.6.1

ELK 三个软件的版本是需要匹配的，如果不知道怎么匹配，就全选最新的版本，这样是没问题的。

11.2.1 安装配置启动 Redis

第一步，在开发机下载 redis-3.2.6.tar.gz，下载地址为：<https://redis.io/>。

第二步，将在开发机下载好的 redis-3.2.6.tar.gz 复制到 10.211.55.4。

```
scp redis-3.2.6.tar.gz root@10.211.55.4:/opt/
```

第三步，解压安装。

```
cd /opt/  
tar -zxvf redis-3.2.6.tar.gz  
cd redis-3.2.6/  
make && make install
```

第四步，修改配置文件。

```
vi /opt/redis-3.2.6/redis.conf
```

修改如下配置：

```
bind 10.211.55.4
```

第五步，启动 redis-server。

```
redis-server /opt/redis-3.2.5/redis.conf
```

之后，通过 rdm 连接或 redis-cli 操作试试是否启动成功。

11.2.2 安装配置启动 Elasticsearch

第一步，在开发机下载 elasticsearch-2.4.0.tar.gz，下载地址为：<https://www.elastic.co/downloads>。

第二步，将在开发机下载好的 elasticsearch-2.4.0.tar.gz 复制到 10.211.55.4。

```
scp elasticsearch-2.4.0.tar.gz root@10.211.55.4:/opt/
```

第三步，解压安装。

```
cd /opt/  
tar -zxvf elasticsearch-2.4.0.tar.gz
```


第四步，修改配置文件。

```
vi /opt/elasticsearch-2.4.0/config/elasticsearch.yml
```

修改如下配置：

```
cluster.name: mymicroservice-elk
node.name: node-es-1
path.data: /data/es/data
path.logs: /data/es/logs
network.host: 10.211.55.4
http.port: 9200
```

这里，指定了该 Elasticsearch 节点所属的集群、节点的名称、数据文件存储的位置、日志文件存储的位置，以及该节点绑定的 host 和 port。

第五步，启动 Elasticsearch。

```
sudo chown -R xiaoming:xiaoming /opt/elasticsearch-2.4.0/
sudo chown -R xiaoming:xiaoming /data/es/
nohup /opt/elasticsearch-2.4.0/bin/elasticsearch &
```

注意，Elasticsearch 不允许以 root 用户来启动服务，所以需要先把与 Elasticsearch 相关的文件夹的操作权限赋给非 root 用户（这里是 xiaoming 组中的 xiaoming 用户），之后以 xiaoming 用户启动 Elasticsearch 服务。

第六步，测试 Elasticsearch。

在浏览器中输入“http://10.211.55.4:9200/”进行访问或者在服务器上使用：

```
curl -X GET "http://10.211.55.4:9200"
```

访问 Elasticsearch，返回如下结果，表示 Elasticsearch 安装成功！

```
{
  "name" : "node-es-1",
  "cluster_name" : "mymicroservice-elk",
  "version" : {
    "number" : "2.4.0",
    "build_hash" : "ce9f0c7394dee074091dd1bc4e9469251181fc55",
    "build_timestamp" : "2016-08-29T09:14:17Z",
    "build_snapshot" : false,
```

```

"lucene_version" : "5.5.2"
},
"tagline" : "You Know, for Search"
}

```

11.2.3 安装配置启动 Logstash-Shipper

第一步，在开发机下载 logstash-2.4.0.tar.gz，下载地址为：<https://www.elastic.co/downloads>。

第二步，将在开发机下载好的 logstash-2.4.0.tar.gz 复制到 10.211.55.4。

```
scp logstash-2.4.0.tar.gz root@10.211.55.4:/opt/
```

第三步，解压安装。

```
cd /opt/
tar -zxvf logstash-2.4.0.tar.gz

```

第四步，重命名解压好的包。

```
mv /opt/logstash-2.4.0/ /opt/logstash-shipper
```

第五步，创建并添加配置文件。

```
cd /opt/logstash-shipper/
mkdir conf
cd conf/
touch logstash.conf
vi logstash.conf

```

添加如下配置：

```

input {
  tcp {
    mode => "server"
    host => "10.211.55.4"
    port => 4560
    codec => "json_lines"
  }
}

filter {

```

```

}

output {
  redis{
    data_type => "list"
    host => ["10.211.55.4:6379"]
    key => "microservice:logstash:redis"
  }
}

```

这里，配置了 Logstash 的三要素：input、filter 和 output。input 指定数据从哪里来，filter 对数据进行过滤处理，output 指定将处理后的数据发送到哪里去。Logstash 提供了很多的 input、filter 和 output 插件。

这里使用的 input 插件是 tcp 插件。

- mode: 可选 server 或 client。server 表示监听客户端连接；client 表示去连接 server。
- host: 监听的 host
- port: 监听的端口
- codec: 对输入数据进行编解码，转换格式，这样，就不需要在 filter 中做这个事儿了。“json_lines”是对有换行符（“\n”）的 json 串进行编解码，“json”是对没有换行符的 json 串进行编解码。

使用的输出插件是 Redis 插件。

- data_type: 可选 list、channel 或 pattern_channel。这里使用了 list，之后使用 blpop 命令处理 Redis 内的元素。blpop 是 lpop 的阻塞版本，当给定列表内没有任何元素可供弹出的时候，连接将被 blpop 命令所阻塞。
- host: Redis server 的地址。
- key: 这里是 list 的名字。

第六步，启动 Logstash-Shipper。

```

nohup /opt/logstash-shipper/bin/logstash -f /opt/logstash-shipper/conf/
logstash.conf &

```

这里使用“-f”指定启动的时候读取的配置文件。

11.2.4 安装配置启动 Logstash-Indexer

第一步，解压安装。

```
cd /opt/
tar -zxf logstash-2.4.0.tar.gz
```

第二步，重命名解压好的包。

```
mv /opt/logstash-2.4.0/ /opt/logstash-indexer
```

第三步，创建并添加配置文件。

```
cd /opt/logstash-indexer/
mkdir conf
cd conf/
touch logstash.conf
vi logstash.conf
```

添加如下配置：

```
input {
  redis{
    data_type => "list"
    host => "10.211.55.4"
    port => 6379
    key => "microservice:logstash:redis"
  }
}

filter {
}

output {
  elasticsearch {
    hosts => ["10.211.55.4:9200"]
    index => "logstash-%{+YYYY.MM.dd}"
  }
}
```

这里，同样配置了 Logstash 的三要素：input、filter 和 output。使用的 input 插件是 Redis 插件，各个选项意义与 Shipper 中的相同。

使用的输出插件是 Elasticsearch 插件。

- host: Elasticsearch 集群的地址，如果有多个节点，节点之间使用“,”分隔。注意，在该集群中，最好只配置 Elasticsearch 的 client 节点。
- index: 指定数据发到哪一个索引中去。默认使用“logstash-%{+YYYY.MM.dd}”做索引。
 - 按照天来分片，方便我们按照天来查询和删除存储的日志数据。
 - 在语法解析的时候，看到以+号开头的，会自动认为后面是时间格式，尝试用时间格式来解析后续字符串。所以，不要给自定义的字段起一个以+号开头的名字。
 - 索引名中不能有大写字母。
 - 有时也会自定义为 logstash-%{servicename}-%{+YYYY.MM.dd}，在索引中添加服务名。

第四步，启动 Logstash-Indexer。

```
nohup /opt/logstash-indexer/bin/logstash -f /opt/logstash-indexer/conf/
logstash.conf &
```

11.2.5 安装配置启动 Kibana

第一步，在开发机下载 kibana-4.6.1-linux-x86_64.tar.gz，下载地址为：<https://www.elastic.co/downloads>。

第二步，将在开发机下载好的 kibana-4.6.1-linux-x86_64.tar.gz 复制到 10.211.55.4。

```
scp kibana-4.6.1-linux-x86_64.tar.gz root@10.211.55.4:/opt/
```

第三步，解压安装。

```
cd /opt/
tar -zxvf kibana-4.6.1-linux-x86_64.tar.gz
```

第四步，修改配置文件。

```
vi /opt/kibana-4.6.1-linux-x86_64/config/kibana.yml
```


修改如下配置：

```
server.host: "10.211.55.4"
server.port: 5601
elasticsearch.url: http://10.211.55.4:9200
```

这里，server.host 和 server.port 指定了 Kibana 的地址；elasticsearch.url 指定了 Kibana 从哪些 Elasticsearch 节点中查询数据。

第五步，启动 Kibana。

```
nohup /opt/kibana-4.6.1-linux-x86_64/bin/kibana &
```

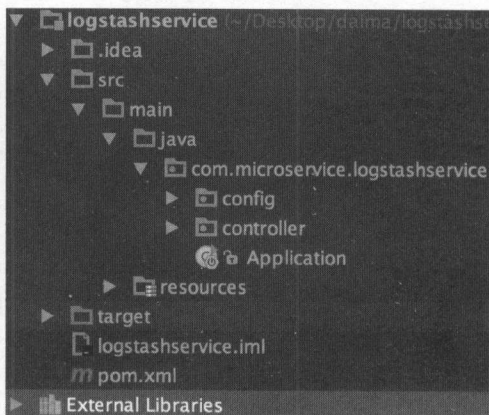
第六步，测试 Kibana。

在浏览器中输入“http://10.211.55.4:5601/”进行访问。当出现 Kibana 的页面时，表示 Kibana 启动成功！

至此，我们就完成了整套 ELK 缓冲架构的搭建!!!

11.3 使用 LogbackAppender 发送日志

SpringBoot 默认选择 Logback 作为日志框架。所以这里我们准备使用 LogbackAppender 通过 tcp 协议来发送日志到 ELK 系统中。为什么不选择直接从日志文件提取数据到 Logstash-Shipper 的方式来收集日志呢？因为这种方式需要在每一个服务器上都安装一个 Logstash，比较麻烦，而使用 tcp 协议是不需要安装的；但是使用 tcp 协议需要写一些代码，看似对于每个服务都要写这么一段代码，而实际上，每个公司基本都会有自己的服务框架，把日志发送的这段代码写在其中，其他服务依赖于这个服务框架，这样写一遍就可以了。



11.3.1 搭建项目框架

新建一个服务 logstashservice，其项目结构如图 11-3 所示。

图11-3 logstashservice项目结构

其中，pom.xml 文件内容如下：

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.
apache.org/maven-v4_0_0.xsd">
    <modelVersion>4.0.0</modelVersion>

    <groupId>com.microservice</groupId>
    <artifactId>logstashservice</artifactId>
    <version>1.0-SNAPSHOT</version>

    <properties>
        <java.version>1.8</java.version>
    </properties>
    <parent>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-parent</artifactId>
        <version>1.4.3.RELEASE</version>
    </parent>

    <dependencies>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-web</artifactId>
        </dependency>
        <dependency>
            <groupId>io.springfox</groupId>
            <artifactId>springfox-swagger2</artifactId>
            <version>2.2.2</version>
        </dependency>
        <dependency>
            <groupId>io.springfox</groupId>
            <artifactId>springfox-swagger-ui</artifactId>
            <version>2.2.2</version>
        </dependency>
        <!-- logstash-logback -->
        <dependency>
            <groupId>net.logstash.logback</groupId>
```

```

        <artifactId>logstash-logback-encoder</artifactId>
        <version>4.6</version>
    </dependency>
</dependencies>

<build>
    <plugins>
        <plugin>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-maven-plugin</artifactId>
        </plugin>
    </plugins>
</build>
</project>

```

除了熟悉的 `spring-boot-starter-web` 与 `Swagger` 依赖外，还引入了 `logstash-logback-encoder` 依赖。

引入依赖之后，创建服务启动主类，代码如下：

```

package com.microservice.logstashservice;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import springfox.documentation.swagger2.annotations.EnableSwagger2;

@SpringBootApplication
@EnableSwagger2
public class Application {
    public static void main(String[] args) {
        SpringApplication sa = new SpringApplication(Application.class);
        sa.run(args);
    }
}

```

依然是熟悉的启动主类。

11.3.2 配置 logback.xml 文件

通常，Logback 的日志配置信息都会存放在 `logback.xml` 文件中，在 `SpringBoot` 中配置

该文件非常简单。配置如下：

```
<?xml version="1.0" encoding="UTF-8"?>
<configuration>
<include resource="org/springframework/boot/logging/logback/base.xml"/>
<logger name="org.springframework.web" level="DEBUG"/>
</configuration>
```

Spring Boot 默认选择 Logback 作为日志框架。在非 Web 应用中，使用日志框架，需要引入 spring-boot-starter-logging，在 Web 项目中，引入 spring-boot-starter-web 就可以了。如果我们只是想改变某些日志的 level，那么在 logback.xml 文件中，一般会引入 org/springframework/boot/logging/logback/base.xml（该文件在 <https://github.com/spring-projects/spring-boot/blob/master/spring-boot/src/main/resources/org/springframework/boot/logging/logback/base.xml> 上），在该文件中，配置了很多信息，例如 Tomcat 的一些日志信息。引入之后，定义一些日志的级别就可以了，比如上边的<logger name="org.springframework.web" level="DEBUG"/>。

11.3.3 创建 LogbackAppender 发送日志

在使用 LogbackAppender 向 Logstash-Shipper 发送日志信息之前，首先需要配置 Logstash-Shipper 和日志级别等相关信息，这些信息配置在 application.properties 中，内容如下：

```
logstash.shipper.host=10.211.55.4
logstash.shipper.port=4560
logstash.level=info
```

这里配置了 Logstash-Shipper 的地址及日志级别（为 info）。只有日志级别大于等于 info 的日志（例如，info、warn 和 error）才会被发送到 Logstash，而 debug 级别的就不发送了。

配置好 Logstash 和日志级别之后，创建一个 LogbackAppender 来发送日志到 Logstash-Shipper。代码如下：

```
package com.microservice.logstashservice.config;

import ch.qos.logback.classic.Level;
import ch.qos.logback.classic.Logger;
import net.logstash.logback.appender.LogstashTcpSocketAppender;
import net.logstash.logback.encoder.LogstashEncoder;
```



```

import org.slf4j.LoggerFactory;
import org.springframework.beans.factory.annotation.Value;
import org.springframework.stereotype.Component;

import javax.annotation.PostConstruct;
import java.net.InetSocketAddress;

@Component
public class LogstashConfig {
    @Value("${logstash.shipper.host}")
    private String shipperHost;
    @Value("${logstash.shipper.port}")
    private int shipperPort;
    @Value("${logstash.level}")
    private String logLevel;

    @PostConstruct
    public void startLogbackAppender() {
        Logger rootLogger = (Logger) LoggerFactory.getLogger(Logger.ROOT_LOGGER_NAME);
        LogstashTcpSocketAppender appender = new LogstashTcpSocketAppender();
        //tcp appender
        appender.setName("stash");
        appender.addDestinations(new InetSocketAddress(shipperHost, shipperPort));

        LogstashEncoder encoder = new LogstashEncoder();
        String servicename = "logstashservice";
        encoder.setCustomFields("{ \"service\": \"" + servicename + "\" }"); //服务名会在日志中显示（可以方便地知道该日志是哪个服务的）
        encoder.start();

        appender.setEncoder(encoder);
        appender.setContext(rootLogger.getLoggerContext());
        appender.start();
        rootLogger.addAppender(appender);
        rootLogger.setLevel(Level.toLevel(logLevel));
    }
}

```

该类有几个需要注意的地方：首先，使用@PostConstruct 注解指定 startLogbackAppender()

方法在该类实例化之后就开始执行；其次，tcpAppender 内部使用了 `disruptor` 框架，`disruptor` 是一个优秀的无锁框架，并且在该框架中日志的发送是异步的，所以不需要我们在程序上做异步处理；最后，想要理清 `startLogbackAppender()` 方法中的各个实例的层次结构，可以参考 Logstash 的 GitHub 上的一段配置：

```
<?xml version="1.0" encoding="UTF-8"?>
<configuration>
  <appender name="stash"
    class="net.logstash.logback.appender.LogstashTcpSocketAppender">
    <destination>127.0.0.1:4560</destination>
    <!-- encoder is required -->
    <encoder class="net.logstash.logback.encoder.LogstashEncoder" />
  </appender>
  <root level="DEBUG">
    <appender-ref ref="stash" />
  </root>
</configuration>
```

最后，总结一下 `startLogbackAppender()` 的流程。首先定义一个 `LogstashTcpSocketAppender` 的实例，名字为 `stash`，发送到的目的地为指定的 `Logstash-Shipper` 的地址；之后为了将数据输出为 json 串，创建了 `LogstashEncoder` 实例，并且指定了自定义的字段，这样就可以在 Kibana 中使用 “service: 'logstashservice'” 来查询日志。然后，将该 `encoder` 实例赋给 `stash`。最后，将 `stash` 赋给 Logback 的 `rootLog`，并指定记录日志的级别。

至此，核心代码就完成了！下面，创建 `controller` 进行测试。

11.3.4 验证日志输出查询功能

代码如下：

```
package com.microservice.logstashservice.controller;

import io.swagger.annotations.Api;
import io.swagger.annotations.ApiOperation;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.web.bind.annotation.*;

@Api("logstash 相关 API")
@RestController
```

```

@RequestMapping("/logstash")
public class TestController {
    private static final Logger LOGGER = LoggerFactory.getLogger(TestController.class);

    @ApiOperation("测试 logstash")
    @RequestMapping(value = "/test/{username}", method = RequestMethod.GET)
    public int getUserAge(@PathVariable("username") String username) {
        int age = 0;
        LOGGER.info("start: getUserAge, username:'{}'", username);
        if (username.equals("小红")) {
            age = 18;
            LOGGER.debug("end: getUserAge, username:'{}', age:'{}'", username, age);
        } else {
            age = 20;
            LOGGER.error("end: getUserAge, username:'{}', age:'{}'", username, age);
        }
        return age;
    }
}

```

代码编写完成之后, 使用 Swagger 进行测试。在 Kibana 上查询日志。在我们第一次打开 Kibana 时 (<http://10.211.55.4:5601/>), 需要先创建索引。如图 11-4 所示。

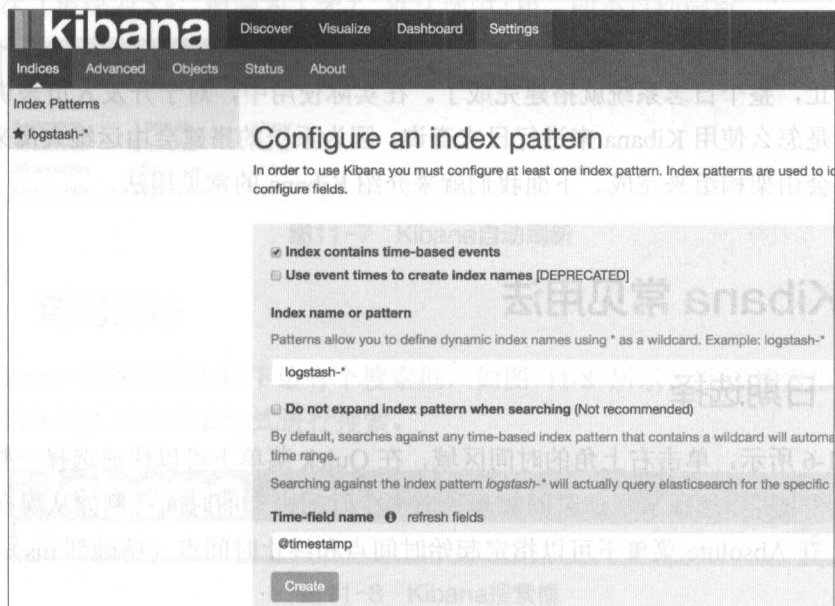


图11-4 创建索引

在 Settings 菜单下创建，因为我们在 Logstash-Indexer 中指定索引是“logstash-%{+YYYY.MM.dd}”，所以这里需要在 Index name or pattern 文本框中输入“logstash-*”，其他选项使用默认值，单击 Create 按钮，索引就创建成功了。创建好索引后，单击 Discover 菜单，在左侧选择相应的索引，之后就可以查看日志了，如图 11-5 所示。

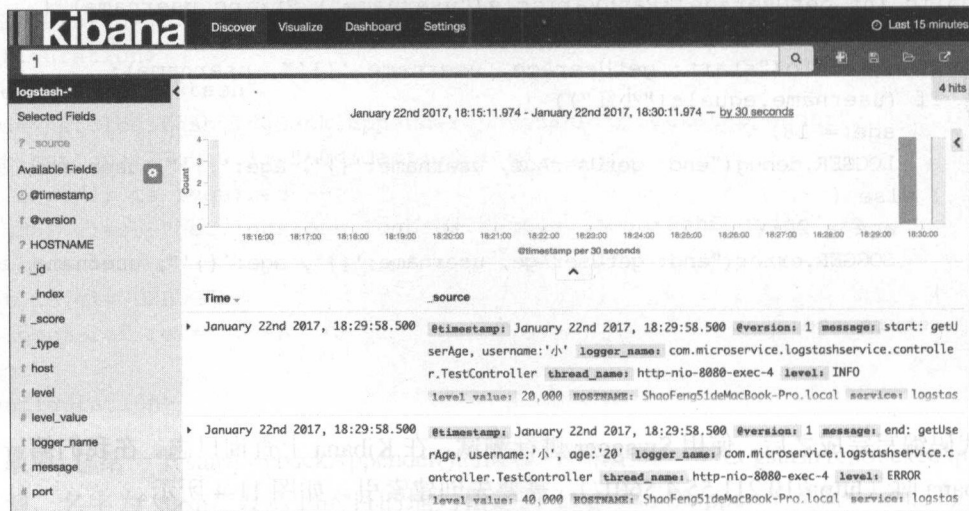


图11-5 Kibana日志查询

到此为止，整个日志系统就搭建完成了。在实际使用中，对于开发人员来讲，可能更需要关注的是怎么使用 Kibana 来进行日志查询，因为系统的搭建会由运维人员来完成，日志发送代码会由架构组来完成。下面我们就来介绍 Kibana 的常见用法。

11.4 Kibana 常见用法

11.4.1 日期选择

如图 11-6 所示，单击右上角的时间区域，在 Quick 菜单下可以快速选择一些时间；在 Relative 菜单下可以选择从当前时间算起“前多少时间内”的时间（例如从现在算起，前 30min 内）；在 Absolute 菜单下可以指定起始时间点和终止时间点（精确到 ms）。

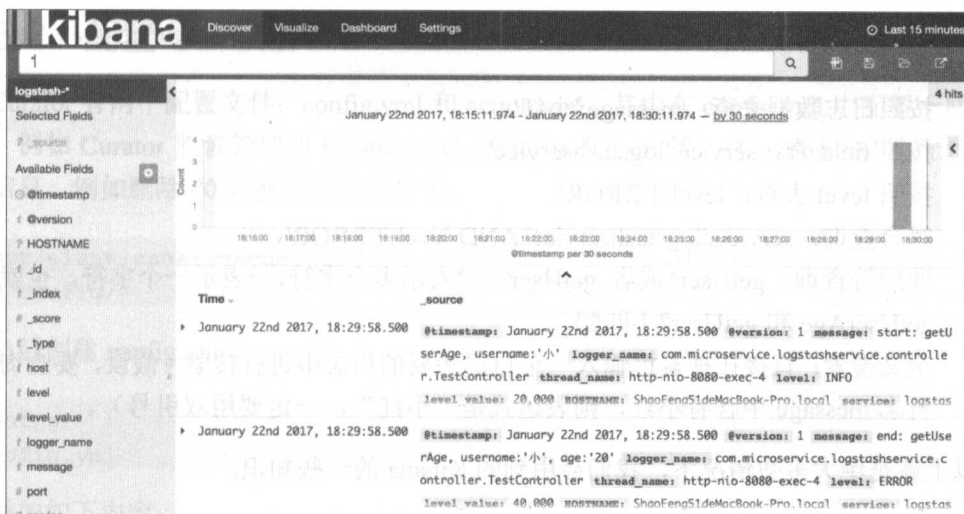


图11-6 Kibana日期选择

11.4.2 自动刷新

如图 11-7 所示，单击右上角的时间区域后，单击 Auto-refresh 菜单。可以选择多长时间刷新一次（最快为 5 s，最慢为 1 天），默认选中 Off，即不自动刷新。

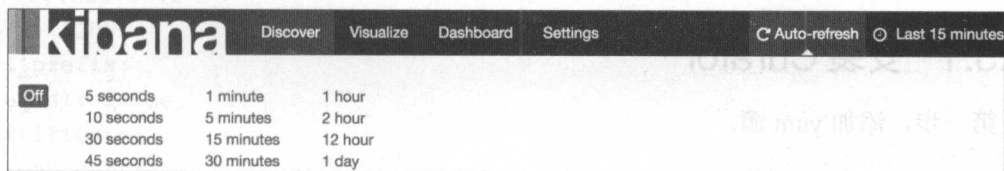


图11-7 Kibana自动刷新

11.4.3 查询语法

在 Kibana 的顶部菜单栏下边有个搜索框，如图 11-8 所示，在该搜索框内可以根据 Kibana 的语法输入相关的表达式进行搜索。

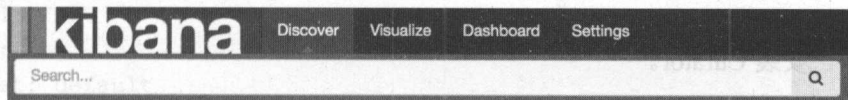


图11-8 Kibana搜索框

值得注意的是，在 Chrome 浏览器下，该搜索框有的时候不好用。所以使用 Kibana 时

推荐换一个浏览器，例如 Safari。我们来看一下 Kibana 最常用的几个查询语法。

- 按照日志数据去查: `message:'小红'`。
- 按照 field 查: `service:'logstashservice'`。
- 按照 level 去查: `level:'ERROR'`。
- 组合查询: `service:'logstashservice' AND level:'ERROR'`。
- 通配符查询: `getUser*`或者 `getUser?`。`*`表示多个字符，`?`表示一个字符，也就是说 `getUserAge` 和 `getUser?`不匹配。
- 全文搜索: 直接在搜索框输入一个自己想要的日志串进行搜索（假设，要查询一个日志 `message` 中含有小红，则表达式是“小红”，一定要用双引号）。

以上就是绝大多数情况下，我们会用到的 Kibana 的一些知识。

11.5 再学一招：使用 Curator 定时删除日志

Elasticsearch-Curator 是一个管理和监督 Elasticsearch 的工具。当日志数据变得很庞大时，我们需要删除一些比较老的数据，以便释放一些硬盘空间，这个时候可以使用 Curator。下面我们就来看一下 Curator 的使用方法。

11.5.1 安装 Curator

第一步，添加 yum 源。

```
tee /etc/yum.repos.d/curator.repo <<-'EOF'
[curator-4]
name=CentOS/RHEL 7 repository for Elasticsearch Curator 4.x packages
baseurl=http://packages.elastic.co/curator/4/centos/7
gpgcheck=1
gpgkey=http://packages.elastic.co/GPG-KEY-elasticsearch
enabled=1
EOF
```

第二步，安装 Curator。

```
cd /opt/
yum install elasticsearch-curator
```


11.5.2 配置 Curator

Curator 有两个配置文件: config.yml 和 action.yml。其中在 config.yml 中配置一些通用信息,例如 Curator 监督管理的 Elasticsearch 的 hosts 和 port 等;在 action.yml 中配置一些动作信息,例如删除 60 天前的日志文件等。

```
cd /opt/elasticsearch-curator/  
mkdir config/
```

之后创建 config.yml。

```
cd config/  
vi config.yml
```

添加如下内容:

```
---  
# Remember, leave a key empty if there is no value. None will be a string,  
# not a Python "NoneType"  
client:  
  hosts:  
    - 10.211.55.4  
  port: 9200  
  url_prefix:  
  use_ssl: False  
  certificate:  
  client_cert:  
  client_key:  
  ssl_no_validate: False  
  http_auth:  
  timeout: 30  
  master_only: False  
  
logging:  
  loglevel: INFO  
  logfile:  
  logformat: default  
  blacklist: ['elasticsearch', 'urllib3']
```

在该配置文件中,比较重要的是 Elasticsearch 的 hosts 和 port,其他采用默认值即可(注

意,如果没有值,需要置空)。该配置文件可以到 Elasticsearch 的官网: <https://www.elastic.co/guide/en/elasticsearch/client/curator/4.1/configfile.html> 下载。

之后在/opt/elasticsearch-curator/config 创建 action.yml。

```
vi action.yml
```

添加如下内容:

```
---
# Remember, leave a key empty if there is no value. None will be a string,
# not a Python "NoneType"
#
# Also remember that all examples have 'disable_action' set to True. If you
# want to use this action as a template, be sure to set this to False after
# copying it.
actions:
  1:
    action: delete_indices
    description: "Delete indices older than 1 days (based on index name), for
logstash- prefixed indices.
        Ignore the error if the filter does not result in an actionable
list of indices (ignore_empty_list) and exit cleanly"
    options:
      timeout_override:
      continue_if_exception: False
      disable_action: False
    filters:
      - filtertype: pattern
        kind: prefix
        value: logstash-
        exclude:
      - filtertype: age
        source: name
        direction: older
        timestring: '%Y.%m.%d'
        unit: days
        unit_count: 1
        exclude:
```

在该配置文件中，只有一个 action，用于删除索引。这里指定了删除前缀是“logstash-”并且日期是一天前的索引。该配置文件也可以到 Elasticsearch 的官网：<https://www.elastic.co/guide/en/elasticsearch/client/curator/4.1/actionfile.html> 下载。

11.5.3 配置 crontab 定时任务

首先编写一个 Shell 脚本：

```
cd /opt/elasticsearch-curator
vi curator-delete-index.sh
```

添加如下内容：

```
#!/bin/sh
echo "start delete data"
/opt/elasticsearch-curator/curator --config /opt/elasticsearch-curator/config/
config.yml /opt/elasticsearch-curator/config/action.yml
echo "delete data sucess"
```

在该 Shell 脚本中，使用 Curator 删除索引。然后，给该脚本赋予执行权限。

```
chmod 777 /opt/elasticsearch-curator/curator-delete-index.sh
```

最后，创建定时任务来执行该脚本。

```
crontab -e
```

打开 vi，输入：

```
20 21 * * * /opt/elasticsearch-curator/curator-delete-index.sh
```

该定时任务指定每天的 21 点 20 分执行脚本。

11.5.4 验证定时任务

笔者在 2017-01-21 的时候访问过服务 logstashservice，在 2017-01-22 的时候也访问过，所以在 /data/es/data/mymicroservice-elk/nodes/0/indices 目录下，会看到两个文件夹，如图 11-9 所示。

```

[centos-linux-2 config]$ cd /data/es/data/mymicroservice-elk/nodes/0/indices
[centos-linux-2 indices]$ ls
logstash-2017.01.21  logstash-2017.01.22

```

图11-9 索引文件

过了 2017-01-22 的 21 点 20 分后, logstash-2017.01.21 文件被删除, 如图 11-10 所示。

```

[centos-linux-2 log]$ cd /data/es/data/mymicroservice-elk/nodes/0/indices
[centos-linux-2 indices]$ ls
logstash-2017.01.22

```

图11-10 删除索引文件

证明 Curator 配置成功!!!

当然, 我们也可以通过 /var/spool/mail/root 文件来查看定时任务的执行日志。如果内容类似下面这样, 则表明配置成功!!!

```

start delete data
2017-01-22 21:20:02,235 INFO      Preparing Action ID: 1, "delete_indices"
2017-01-22 21:20:02,248 INFO      Trying Action ID: 1, "delete_indices": Delete
indices older than 1 days (based on index name), for logstash- prefixed indices.
Ignore the error if the filter does not result in an actionable list of indices
(ignore_empty_list) and exit cleanly
2017-01-22 21:20:02,357 INFO      Deleting selected indices:
['logstash-2017.01.21']
2017-01-22 21:20:02,357 INFO      ---deleting index logstash-2017.01.21
2017-01-22 21:20:02,490 INFO      Action ID: 1, "delete_indices" completed.
2017-01-22 21:20:02,490 INFO      Job completed.
delete data success

```

在该日志中, 可以清楚地看到删除了 logstash-2017.01.21 索引文件。



第 12 章

微服务全链路追踪系统

12.1 初识 Zipkin

Zipkin 是一个用于在分布式系统中实现全链路追踪的工具，尤其是在微服务架构中，Zipkin 可用于收集时间数据以及查找依赖服务，进而查找错误和调优性能。

12.1.1 为什么要使用 Zipkin

来看两个场景。

场景一：假设企业中有成百上千个微服务，用户向服务 A 发出一个请求时，出错了。我们想找出出错的原因，通过查看日志，发现出错的源头不是服务 A，而是服务 A 所调用的服务，但是没有指明是哪一个服务。这个时候可能就陷入了僵局，或者需要去分析配置文件来查找。如果服务 A 调用了多个服务，那么出错的是哪一个呢？假设这些被调用的服务也没有出错，而是它们所调用的服务出错了呢？这又该如何分析？

场景二：假设有 4 个服务，分别是 myservice1、myservice2、myservice3 和 myservice4。调用关系如图 12-1 所示。

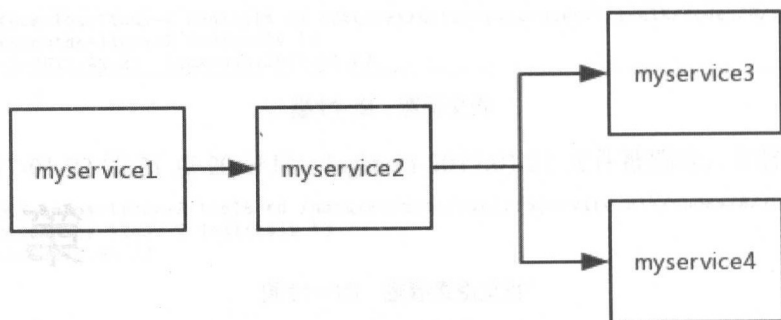


图12-1 服务调用关系

用户向 myservice1 发出一个请求，myservice1 需要调用 myservice2，myservice2 需要调用 myservice3 和 myservice4，最后将结果返回给用户。最后发现，用户的这个请求花费了很长时间甚至还会出现超时错误。我们现在要处理这个问题，首先需要查明一件事：在这条调用链中，到底是哪一个服务慢了呢？如果我们为每个服务的 controller 都通过日志记录了执行时长，这个时候可以通过分别查看每个服务的日志看出一些端倪。但是这是很费劲的，甚至在高并发或者系统具有复杂的调用关系时，这样的方式行不通。怎么办？

为了处理这些问题，Twitter 公司推出了一件全链路追踪的利器：Zipkin!!!（实际上是 GoogleDapper 论文的完成者之一）。下面我们就来看看 Zipkin 是怎样解决这些问题的。

12.1.2 Zipkin 工作流程

首先来看一下 Zipkin 官网 (<http://zipkin.io/>) 提供的一张图。

如图 12-2 所示，首先通过 Reporter 从服务中采集数据，之后通过 Transport 传递给 Zipkin-collector，Zipkin-collector 将日志收集过来之后，通过 Zipkin-storage 存储到 Database 中。之后用户通过 Zipkin-ui 来查询数据，Zipkin-ui 调用相应的 Zipkin-api 从 Zipkin-storage 中查询数据。

这里有几个概念说一下。

- **Reporter:** 在服务中进行数据采集的工具，实际上就是 Zipkin 的各种语言的实现。其中 Java 语言的实现最多，从目前来看官方提供了一种，称为 Brave。各大社区提供了 5 种，其中比较有名的是 spring-cloud-sleuth。我们在之后的代码实现中将使用最为正统的 Brave 来实现全链路追踪。
- **Transport:** 主要完成将服务传递过来的 span 转换成 Zipkin 通用的 span，并传输给

Zipkin-collector。目前有 3 种最主要的 transporter: http、kafka 及 scribe。笔者在之后的代码实现中将使用最为常用的 http 作为 transporter, 这也是 Zipkin 默认采用的 transporter。

- collector: Zipkin 四大组件之罗网。主要用于对 transporter 传输过来的追踪数据进行验证、存储和设置索引。
- storage: Zipkin 四大组件之仓库。主要用于存储追踪数据。常见的有 4 类: in-memory、MySQL、Cassandra 及 Elasticsearch。其中 in-memory 是默认的, 这种存储直接将数据存储在内存中, Zipkin-server 宕机之后, 数据就丢失了; Cassandra 是 Twitter 公司最推荐的存储仓库。我们将会使用 MySQL 来存储, 因为 MySQL 对于绝大多数读者来讲都是最熟悉的数据库。
- search-api: Zipkin 四大组件之刀锋, 提供查询 api。
- web-ui: Zipkin 四大组件之门面。web-ui 提供了按照服务、时间以及标记(annotations)进行搜索的能力。值得注意的是, Zipkin 的 web-ui 没有提供控制权限的功能。

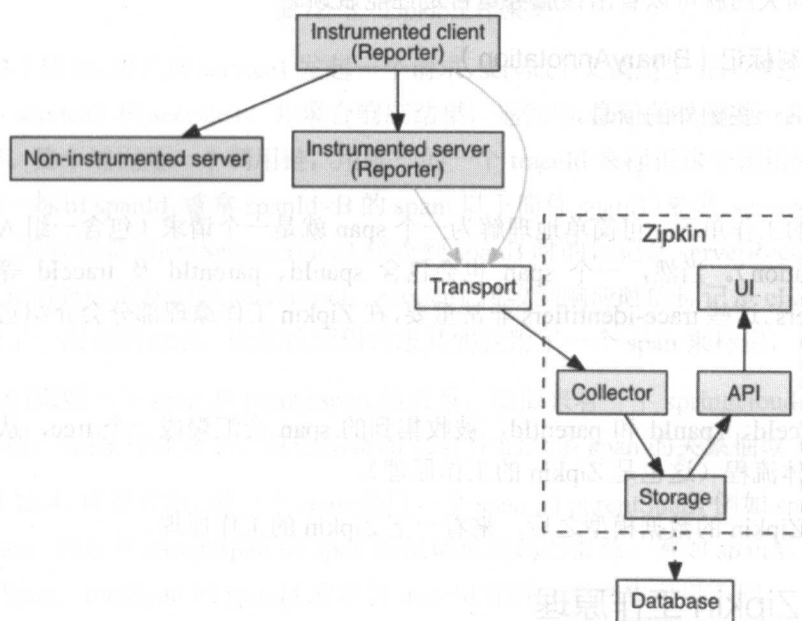


图12-2 Zipkin工作流程图

Zipkin 的整个工作流程我们了解了, 但是 Zipkin 内部是通过怎样的一个机制来追踪调用链呢? 在深入分析 Zipkin 工作原理之前, 首先需要了解一下 Zipkin 的几个数据模型。

12.1.3 Zipkin 数据模型

Zipkin 有 4 种基本的数据模型：Annotation、BinaryAnnotation、span 及 trace。

1. 标记 (Annotation)

标记主要用于定位一个请求的开始和结束，在 Zipkin 中有 4 种标记：cs、sr、ss 和 cr。

- cs (Client Sent)：调用方发起一个请求，这是一个 span 的开始。
- sr (Server Received)：被调方接收到请求，开始处理这个请求。
- ss (Server Sent)：被调方处理完成，并将响应结果返回给调用方。
- cr (Client Received)：调用方接收到被调方的响应，这是一个 span 的结束。

细心的读者可以发现，其实 cr-cs 这个时间就是请求从发出到接收到响应整个流程消耗的时间；sr-cs 及 cr-ss 就是网络延迟的时间；ss-sr 这段时间也就是被调方处理请求的时间，通过这个时间大约就可以看出该服务是否是性能瓶颈。

2. 二进制标记 (BinaryAnnotation)

用于提供一些额外的信息。

span

最基本的工作单元，可简单地理解为一个 span 就是一个请求（包含一组 Annotation 和 BinaryAnnotation）。当然，一个 span 也会包含 spanId、parentId 及 traceId 等比较重要的 trace-identifiers。这些 trace-identifiers 非常重要，在 Zipkin 工作原理部分会介绍它们的作用。

trace

通过 traceId、spanId 和 parentId，被收集到的 span 会汇聚成一个 tree，从而提供一个 request 的整体流程（这也是 Zipkin 的工作原理）。

了解了 Zipkin 的数据模型之后，来看一下 Zipkin 的工作原理。

12.1.4 Zipkin 工作原理

首先来看 springcloud 在 GitHub 上(<https://github.com/spring-cloud/spring-cloud-sleuth>)提供的一张图，如图 12-3 所示。

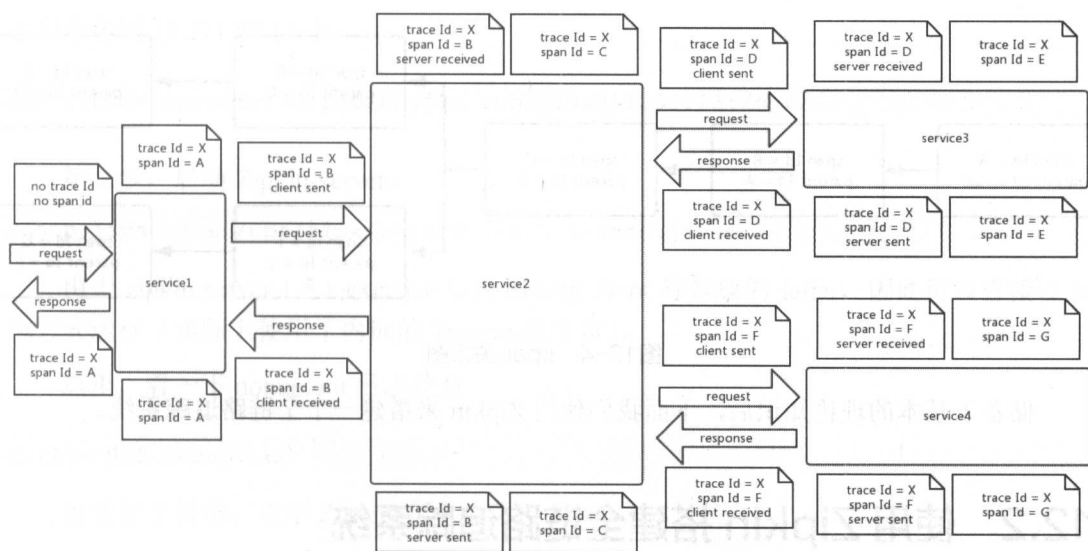


图12-3 Zipkin工作原理

如图 12-3 所示,用户向 service1 发起一个请求,service1 又调用了 service2,然后 service2 分别调用了 service3 和 service4,并聚合响应结果,返回给 service1,最后 service1 将响应返回给用户。整个过程是一个调用链,用唯一的一个 traceId 来标识这个调用链。每个 span 都有一个唯一标识 spanId,就拿 spanId=B 的 span(以下简称 spanB)来讲,service1 向 service2 发出请求时,其标记是 clientSent; service2 接收到 spanB 时的标记是 serverReceived; service2 向 service1 发出响应的标记是 serverSent; service1 接收到响应时的标记是 clientReceived。这样就完成了一次调用请求,而这次调用请求其实就用了一个 span 来标记,就是 spanB。

下边我们来看一下 span 和 parentSpan 的关系,依旧来看一下 springcloud 在 GitHub 上提供的一张图,如图 12-4 所示。这张图将图 12-3 中的众多 span 的关系抽取了出来。

通过图 12-4,可以看出,前一个 span 是后一个 span 的 parentSpan,例如 spanA 是 spanB 的 parentSpan; 而没有 parentSpan 的 span 就是请求发起的根源。例如 spanA,这样的 span 又称为 rootSpan, rootSpan 的 spanId 通常与 traceId 相同,当然也可以不同。

最后,简单总结一下 Zipkin 的工作流程:使用 traceId 贯穿整个调用链,用来标记整个调用链的一次请求;使用 span 来记录单次的服务调用流程,在 span 中使用 cs、sr、ss 和 cr 4 个标记记录 4 个基本事件;使用 parentSpan 与 span 来编排整个调用链的调用次序,并与 traceId 配合将整个调用链编织起来。

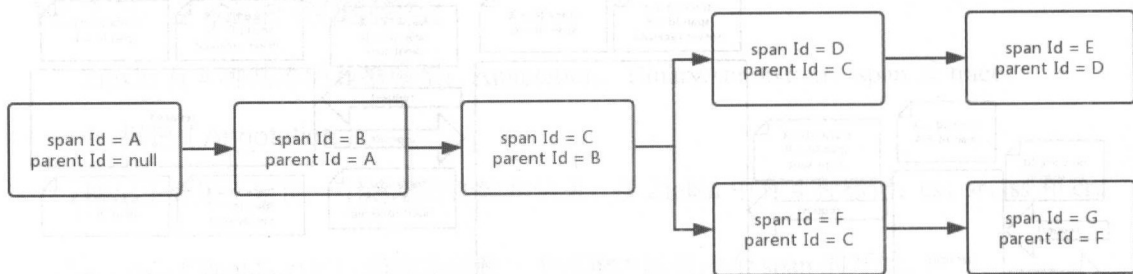


图12-4 span关系图

储备了基本的理论知识后，下面我们使用 Zipkin 来搭建一个全链路追踪系统。

12.2 使用 Zipkin 搭建全链路追踪系统

使用 Zipkin 搭建全链路追踪系统非常简单。常用的搭建方式有两种：一种是使用 Docker 镜像；另一种是使用 jar 包。前者是 Zipkin 官方最推荐的方式，但是需要你具备一点 Docker 的基础知识，比如一些基本的 Docker 命令。关于 Docker 的基本命令，我们在第 13 章讲解，这里使用 jar 包部署，由于 Zipkin 服务是一个 Spring Boot 程序，因而使用 jar 包部署也非常简单。

笔者的环境是：

- 操作系统，centos7；ip 是 10.211.55.13。
- JDK，1.8.0_102（注意，Zipkin-server 需要 JDK8+）。
- Zipkin-server，1.5.1。

第一步，下载 Zipkin-server。

下载 Zipkin-server 有两种方式，第一种是直接在 10.211.55.13 上执行如下语句：

```
wget -O zipkin.jar 'https://search.maven.org/remote_content?g=io.zipkin.  
java&a=zipkin-server&v=1.5.1&c=exec'
```

这种方式要求你的网速比较快，或者你愿意等比较长的时间，比如说一个小时。否则，推荐使用第二种方式。在开发机上先下载好，下载地址是 <http://zipkin.io/pages/quickstart.html>，

之后上传到 10.211.55.13 上。

```
scp zipkin-server-1.5.1-exec.jar root@10.211.55.13:/opt/
```

第二步，启动 Zipkin-server。

```
nohup java -jar /opt/zipkin-server-1.5.1-exec.jar &
```

由于 zipkin-server-1.5.1-exec.jar 是由 Spring Boot 打包成的 fatjar，因此可以直接以 jar 包方式运行（实际上使用了内嵌的 Tomcat 服务器）。

这里，看一下 nohup.out 日志信息。

```
tail -500f nohup.out
```

出现如下日志，表示 Zipkin 启动成功！

```
2017-01-23 18:08:25.990 INFO 7086 --- [main]
s.b.c.e.t.TomcatEmbeddedServletContainer : Tomcat started on port(s): 9411
(http)
2017-01-23 18:08:25.995 INFO 7086 --- [main]
zipkin.server.ZipkinServer : Started ZipkinServer in 5.911 seconds
(JVM running for 6.923)
```

并且可以看到启动的端口是 9411。这就是 Zipkin-server 默认的启动端口。

第三步，在 Zipkin-webUI 上进行查看。

在浏览器中输入 “http://10.211.55.13:9411/”，结果如图 12-5 所示。

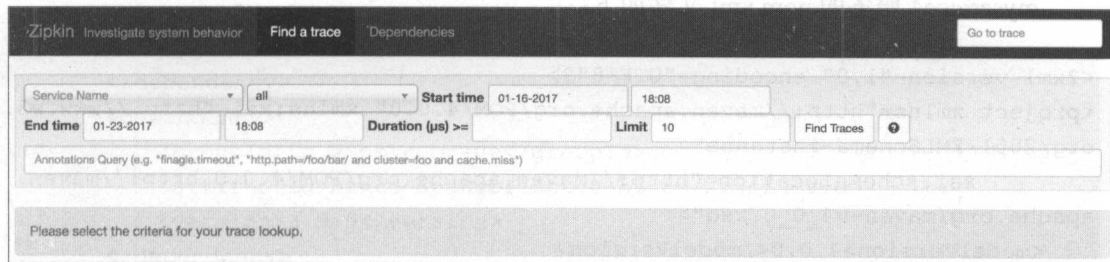


图12-5 Zipkin首页界面

这样，一套基于 Zipkin 的全链路追踪系统就搭建完成了!!!

12.3 使用 Brave + AsyncHttpClient 实现全链路追踪

我们将会建立 4 个微服务：myservice1、myservice2、myservice3 和 myservice4。调用关系如图 12-1 所示。

下面，来实现这 4 个服务。

12.3.1 搭建项目框架

myservice4 项目结构如图 12-6 所示。

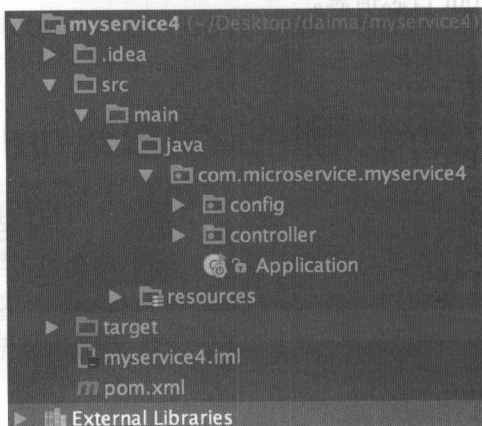


图12-6 myservice4项目结构

其他三个服务的项目框架图与 myservice4 几乎相同，不再贴出。

myservice4 服务的 pom.xml 文件如下：

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.
org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.
apache.org/maven-v4_0_0.xsd">
    <modelVersion>4.0.0</modelVersion>

    <groupId>com.microservice</groupId>
    <artifactId>myservice4</artifactId>
    <version>1.0-SNAPSHOT</version>
```

```
<properties>
  <java.version>1.8</java.version>
</properties>
<parent>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-parent</artifactId>
  <version>1.4.3.RELEASE</version>
</parent>

<dependencies>
  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
  </dependency>
  <dependency>
    <groupId>iq.springfox</groupId>
    <artifactId>springfox-swagger2</artifactId>
    <version>2.2.2</version>
  </dependency>
  <dependency>
    <groupId>io.springfox</groupId>
    <artifactId>springfox-swagger-ui</artifactId>
    <version>2.2.2</version>
  </dependency>
  <!-- zipkin brave -->
  <dependency>
    <groupId>io.zipkin.brave</groupId>
    <artifactId>brave-core</artifactId>
    <version>3.9.0</version>
  </dependency>
  <dependency>
    <groupId>io.zipkin.brave</groupId>
    <artifactId>brave-spancollector-http</artifactId>
    <version>3.9.0</version>
  </dependency>
  <dependency>
    <groupId>io.zipkin.brave</groupId>
    <artifactId>brave-web-servlet-filter</artifactId>
    <version>3.9.0</version>
  </dependency>
```

```

</dependencies>

<build>
  <plugins>
    <plugin>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-maven-plugin</artifactId>
    </plugin>
  </plugins>
</build>
</project>

```

在该 pom.xml 文件中，除了引入了我们熟悉的 spring-boot-starter-web 与 Swagger 依赖之外，还引入了 3 个依赖，其中，brave-core 用于提供 Brave 的核心 API，brave-spancollector-http 用于提供 http 形式的 collector 的相关 API，brave-web-servlet-filter 用于提供 BraveServletFilter 相关的 API，主要用于 serverTrace。

引入依赖之后，创建服务启动主类，代码如下：

```

package com.microservice.myservice4;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import springfox.documentation.swagger2.annotations.EnableSwagger2;

@SpringBootApplication
@EnableSwagger2
public class Application {
    public static void main(String[] args) {
        SpringApplication sa = new SpringApplication(Application.class);
        sa.run(args);
    }
}

```

项目框架搭建完成之后，编写程序来实现全链路追踪功能。

12.3.2 使用服务端拦截器补充追踪信息

为了收集 span 数据，需要创建 span 收集器；为了记录 sr、ss 等事件信息，需要创建 server 拦截器，代码如下：

```

package com.microservice.myservice4.config;

import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;

import com.github.kristofa.brave.Brave;
import com.github.kristofa.brave.EmptySpanCollectorMetricsHandler;
import com.github.kristofa.brave.Sampler;
import com.github.kristofa.brave.SpanCollector;
import com.github.kristofa.brave.http.DefaultSpanNameProvider;
import com.github.kristofa.brave.http.HttpSpanCollector;
import com.github.kristofa.brave.servlet.BraveServletFilter;

@Configuration
public class ZipkinConfig {
    @Bean
    public SpanCollector spanCollector() {
        HttpSpanCollector.Config spanConfig =
            HttpSpanCollector.Config.builder()
                .compressionEnabled(false)
                .connectTimeout(5000)
                .readTimeout(6000)
                .flushInterval(1)
                .build();
        return HttpSpanCollector.create("http://10.211.55.13:9411", spanConfig,
            new EmptySpanCollectorMetricsHandler());
    }

    @Bean
    public Brave brave(SpanCollector spanCollector) {
        Brave.Builder builder = new Brave.Builder("myservice4");//指定
        serviceName
        builder.spanCollector(spanCollector);
        builder.traceSampler(Sampler.create(1));//采集率
        return builder.build();
    }

    @Bean
    public BraveServletFilter braveServletFilter(Brave brave) {

```



```

        return new BraveServletFilter(brave.serverRequestInterceptor(),
                                      brave.serverResponseInterceptor(),
                                      new DefaultSpanNameProvider()); //spanName
    }
}

```

默认使用 httpMethod 的名字, 例如 get, post

在该类中, 创建了 3 个 Bean: spanCollector、brave 和 braveServletFilter。

其中, spanCollector 主要用于接收从 reporter 传过来的 span 数据, 并做相关的调整。首先创建了 HttpSpanCollector.Config, 并设置了几个选项。

- **compressionEnabled**: span 在 transport 之前是否会被 gzipped, 默认是 false。
- **connectTimeout**: 创建与 Zipkin-server 连接的超时时间, 默认是 10 s,
- **readTimeout**: 当与 Zipkin-server 的连接一旦建立好, 马上进行服务输入流操作, 如果输入流在 readTimeout 设置的时间内都没有有效数据传给 Zipkin-server, 则抛出异常, 默认是 60 s。
- **flushInterval**: 在 flushInterval 设置的时间内先将到达的数据存储在一个 BlockingQueue 中, 等到了 flushInterval 设置的时间到达后, collector 将队列中的数据发送到 Zipkin-server, 简单来讲, 就是每隔 flushInterval 设置的时间, collector 向 Zipkin-server 发送一次数据, 默认是 1s。

brave Bean 是最主要的 Bean。在构建该 Bean 时, 指定了 serviceName; 指定了将要使用的 collector 实例; 指定了采样率, 采样率默认是 0.0f~1.0f, 其中 0 表示不采样, 1 表示每次调用都记录, 0.1 表示 10 次调用里记录一次。通常在测试时我们把采样率设置为 1; 在生产环境中, 为了不让这样的边缘功能影响主业务的运转, 采样率通常设得比较低, 比如 0.1。brave Bean 还初始化了 4 个很重要的拦截器: serverRequestInterceptor、serverResponseInterceptor、clientRequestInterceptor 和 clientResponseInterceptor, 分别用于 serverReceived、serverSent、clientSent 和 clientReceived。

braveServletFilter Bean 实现了 javax.servlet.Filter, 其用于 serverReceived 和 serverSent 的处理。

最后, 编写一个被调用接口供 myservice2 使用, 代码如下:

```
package com.microservice.myservice4.controller;
```

```

import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestMethod;
import org.springframework.web.bind.annotation.RestController;

import io.swagger.annotations.Api;

@Api("myservice4 相关 api")
@RestController
@RequestMapping("/myservice4/zipkin")
public class MyService4Controller {
    @RequestMapping(value = "/test", method = RequestMethod.GET)
    public String test() {
        try {
            return "myservice4-response";
        } catch (Exception e) {
            e.printStackTrace();
            return "";
        }
    }
}

```

这样 myservice4 服务就创建完成了。myservice3 服务与 myservice4 几乎相同，不再赘述。myservice2 服务由于需要使用 AsyncHttpClient 来访问，所以除了需要引入 spring-boot-starter-web、Swagger 及 Zipkin 的相关依赖外，还需要引入 async-http-client 依赖，代码如下：

```

<!-- async-http-client -->
<dependency>
    <groupId>com.ning</groupId>
    <artifactId>async-http-client</artifactId>
    <version>1.9.31</version>
</dependency>

```

引入依赖之后，还需要创建 AsyncHttpClient 的 Bean，代码如下：

```

@Bean
public AsyncHttpClient asyncHttpClient() {
    return new AsyncHttpClient();
}

```

与 Zipkin 相关的 3 个 Bean 的创建方式与 myservice4 的几乎相同，不再赘述。

12.3.3 使用客户端拦截器创建、销毁追踪信息

com.microservice.myservice2.controller.Myservice2Controller (重要)

```
package com.microservice.myservice2.controller;

import java.net.URI;
import java.util.concurrent.Future;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestMethod;
import org.springframework.web.bind.annotation.RestController;

import com.github.kristofa.brave.Brave;
import com.github.kristofa.brave.http.DefaultSpanNameProvider;
import com.github.kristofa.brave.http.HttpClientRequest;
import com.github.kristofa.brave.http.HttpClientRequestAdapter;
import com.github.kristofa.brave.http.HttpClientResponseAdapter;
import com.github.kristofa.brave.http.HttpResponse;
import com.ning.http.client.AsyncHttpClient;
import com.ning.http.client.Request;
import com.ning.http.client.RequestBuilder;
import com.ning.http.client.Response;

import io.swagger.annotations.Api;

@Api("myservice2 相关 api")
@RestController
@RequestMapping("/myservice2/zipkin")
public class Myservice2Controller {

    @Autowired
    private AsyncHttpClient asyncHttpClient;

    @Autowired
    private Brave brave;

    @RequestMapping(value = "/test", method = RequestMethod.GET)
```

```

public String test() {
    try {
        /*****service3*****/
        RequestBuilder builder3 = new RequestBuilder();
        String url3 = "http://localhost:8083/myservice3/zipkin/test";
        builder3.setUrl(url3);
        Request request3 = builder3.build();

        clientRequestInterceptor(request3); //cs
        Future<Response> response3 =
asyncHttpClient.executeRequest(request3);
        clientResponseInterceptor(response3.get()); //cr

        /*****service4*****/
        RequestBuilder builder4 = new RequestBuilder();
        String url4 = "http://localhost:8084/myservice4/zipkin/test";
        builder4.setUrl(url4);
        Request request4 = builder4.build();

        clientRequestInterceptor(request4);
        Future<Response> response4 =
asyncHttpClient.executeRequest(request4);
        clientResponseInterceptor(response4.get());

        return response3.get().getResponseBody() + "====" +
response4.get().getResponseBody();
    } catch (Exception e) {
        e.printStackTrace();
        return "";
    }
}

private void clientRequestInterceptor(Request request) {
    brave.clientRequestInterceptor().handle(new
HttpClientRequestAdapter(new HttpClientRequest() {

        @Override
        public URI getUri() {
            return URI.create(request.getUrl());
        }
    }

```



```

@Override
public String getHttpMethod() {
    return request.getMethod();
}

@Override
public void addHeader(String headerKey, String headerValue) {
    request.getHeaders().add(headerKey, headerValue);
}
}, new DefaultSpanNameProvider()));
}

private void clientResponseInterceptor(Response response) {
    brave.clientResponseInterceptor().handle(new
HttpClientResponseAdapter(new HttpResponseMessage() {
        public int getHttpStatusCode() {
            return response.getStatusCode();
        }
    }));
}
}
}

```

该类的 `test()` 方法，使用 `AsyncHttpClient` 请求 `myservice3` 服务，之后请求 `myservice4` 服务，最后将 `myservice3` 和 `myservice4` 的返回值聚合起来返回给调用方。这里的重点是，在真正使用 `asyncHttpClient.executeRequest()` 发起调用之前，首先对 `clientRequestInterceptor` 进行调用，处理了 `clientSent` 事件，创建了一个全新的 `span`（一个 `span` 的开始）；当调用结束后，对 `clientResponseInterceptor` 进行调用，以处理 `clientReceived` 事件，最后，将当前的 `span` 置为 `null`（该处理在本文的源码解析部分会讲）。

到此，`myservice2` 服务也开发完成了。`myservice1` 服务与 `myservice2` 几乎相同，只是在 `controller` 中，只调用了 `myservice2` 的接口。这里就不贴出代码了。

至此，4 个服务就全部开发完成了，下面来验证“链路追踪”功能。

12.3.4 使用 Zipkin-webUI 查询链路追踪信息

首先在 10.211.55.13 上启动 `Zipkin-server`。


```
nohup java -jar /opt/zipkin-server-1.5.1-exec.jar &
```

然后，在本机分别启动 4 个服务。

```
nohup java -jar -Dserver.port=8084 myservice4-1.0-SNAPSHOT.jar &
nohup java -jar -Dserver.port=8083 myservice3-1.0-SNAPSHOT.jar &
nohup java -jar -Dserver.port=8082 myservice2-1.0-SNAPSHOT.jar &
nohup java -jar -Dserver.port=8081 myservice1-1.0-SNAPSHOT.jar &
```

通过 Swagger 访问 myservice1，之后在 Zipkin-webUI (<http://10.211.55.13:9411>) 上进行查看。

首先看一下 trace 页，如图 12-7 所示。这里可以根据 servicename、时间等查询每一次的访问 trace。

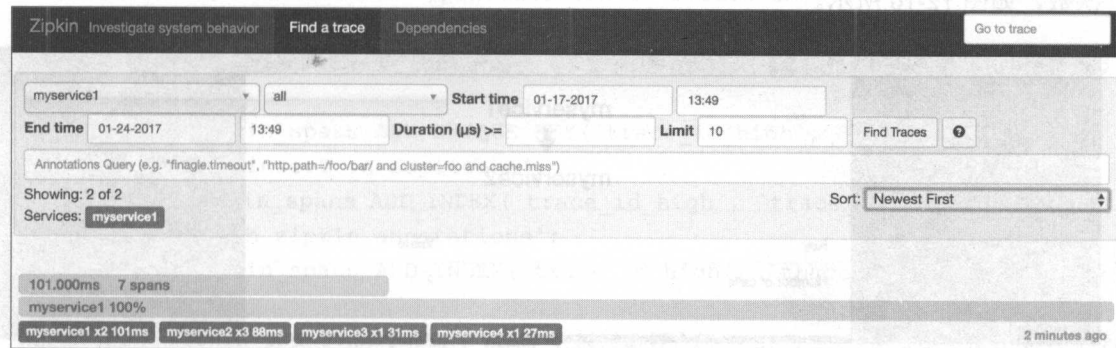


图12-7 trace界面

单击其中的一个 trace 后，还可以看到各个方法的执行时长，如图 12-8 所示，该数据对于寻找性能瓶颈有很大帮助。

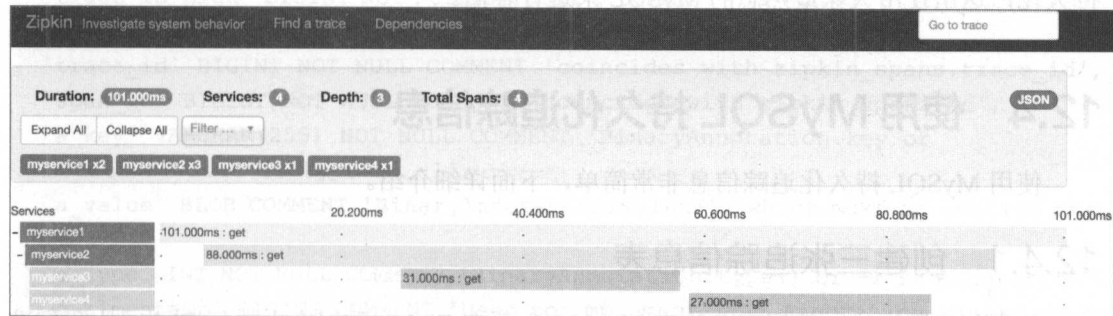


图12-8 trace时间信息

单击界面导航中的 Dependencies，还可以看到依赖关系图，如图 12-9 所示。

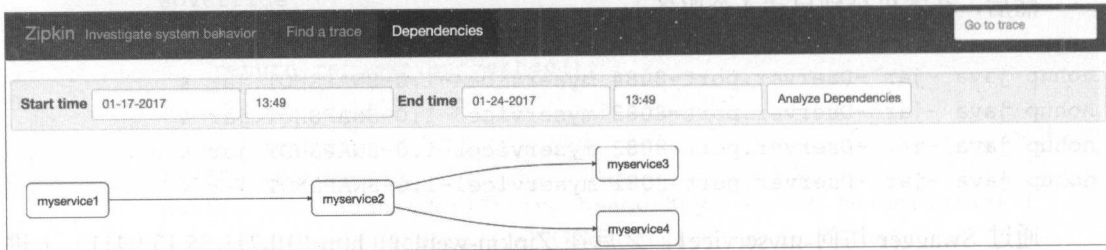


图12-9 服务依赖关系图

有意思的是，服务之间的连线的粗细与调用次数是相关的，调用次数越多，线越粗。调用次数与被调用次数也可以通过单击图中的服务框来看，例如，看 myservice2 的被调用次数，如图 12-10 所示。

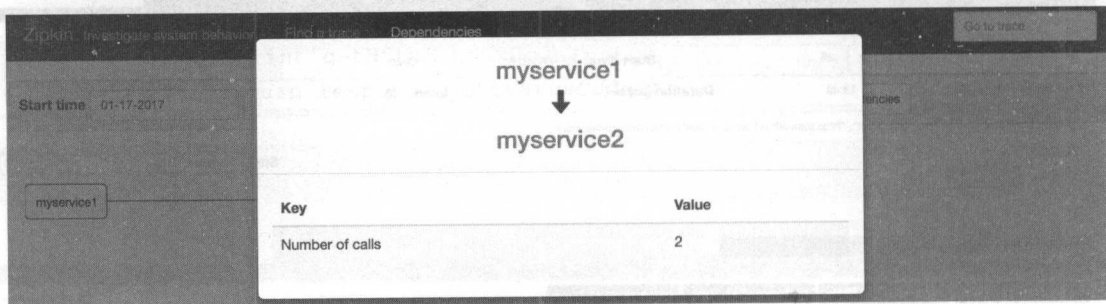


图12-10 服务调用次数

至此，全链路追踪系统就基本完成了，但是现在有一个问题，就是追踪数据是存在内存中的，一旦 Zipkin-server 跑了，之前的数据就都没了。所以，我们需要将追踪数据进行持久化，这里使用大家最熟悉的 MySQL 来进行存储。

12.4 使用 MySQL 持久化追踪信息

使用 MySQL 持久化追踪信息非常简单，下面详细介绍。

12.4.1 创建三张追踪信息表

首先创建数据库，数据库名称为 Zipkin，之后在 Zipkin 数据库中执行 Zipkin 为我们准备好的脚本 mysql.sql 创建三张表及各个索引。

mysql.sql 脚本地址: <https://github.com/openzipkin/zipkin/blob/master/zipkin-storage/mysql/src/main/resources/mysql.sql>。脚本内容:

```
CREATE TABLE IF NOT EXISTS zipkin_spans (
  `trace_id_high` BIGINT NOT NULL DEFAULT 0 COMMENT 'If non zero, this means the
trace uses 128 bit traceIds instead of 64 bit',
  `trace_id` BIGINT NOT NULL,
  `id` BIGINT NOT NULL,
  `name` VARCHAR(255) NOT NULL,
  `parent_id` BIGINT,
  `debug` BIT(1),
  `start_ts` BIGINT COMMENT 'Span.timestamp(): epoch micros used for endTs query
and to implement TTL',
  `duration` BIGINT COMMENT 'Span.duration(): micros used for minDuration and
maxDuration query'
) ENGINE=InnoDB ROW_FORMAT=COMPRESSED CHARACTER SET=utf8 COLLATE utf8_general_ci;

ALTER TABLE zipkin_spans ADD UNIQUE KEY(`trace_id_high`, `trace_id`, `id`)
COMMENT 'ignore insert on duplicate';
ALTER TABLE zipkin_spans ADD INDEX(`trace_id_high`, `trace_id`, `id`) COMMENT
'for joining with zipkin_annotations';
ALTER TABLE zipkin_spans ADD INDEX(`trace_id_high`, `trace_id`) COMMENT 'for
getTracesByIds';
ALTER TABLE zipkin_spans ADD INDEX(`name`) COMMENT 'for getTraces and getSpanNames';
ALTER TABLE zipkin_spans ADD INDEX(`start_ts`) COMMENT 'for getTraces ordering
and range';

CREATE TABLE IF NOT EXISTS zipkin_annotations (
  `trace_id_high` BIGINT NOT NULL DEFAULT 0 COMMENT 'If non zero, this means the
trace uses 128 bit traceIds instead of 64 bit',
  `trace_id` BIGINT NOT NULL COMMENT 'coincides with zipkin_spans.trace_id',
  `span_id` BIGINT NOT NULL COMMENT 'coincides with zipkin_spans.id',
  `a_key` VARCHAR(255) NOT NULL COMMENT 'BinaryAnnotation.key or
Annotation.value if type == -1',
  `a_value` BLOB COMMENT 'BinaryAnnotation.value(), which must be smaller than
64KB',
  `a_type` INT NOT NULL COMMENT 'BinaryAnnotation.type() or -1 if Annotation',
  `a_timestamp` BIGINT COMMENT 'Used to implement TTL; Annotation.timestamp or
zipkin_spans.timestamp',
```

```

`endpoint_ipv4` INT COMMENT 'Null when Binary/Annotation.endpoint is null',
`endpoint_ipv6` BINARY(16) COMMENT 'Null when Binary/Annotation.endpoint is
null, or no IPv6 address',
`endpoint_port` SMALLINT COMMENT 'Null when Binary/Annotation.endpoint is null',
`endpoint_service_name` VARCHAR(255) COMMENT 'Null when Binary/Annotation.
endpoint is null'
) ENGINE=InnoDB ROW_FORMAT=COMPRESSED CHARACTER SET=utf8 COLLATE utf8_general_ci;

ALTER TABLE zipkin_annotations ADD UNIQUE KEY(`trace_id_high`, `trace_id`,
`span_id`, `a_key`, `a_timestamp`) COMMENT 'Ignore insert on duplicate';
ALTER TABLE zipkin_annotations ADD INDEX(`trace_id_high`, `trace_id`, `span_id`)
COMMENT 'for joining with zipkin_spans';
ALTER TABLE zipkin_annotations ADD INDEX(`trace_id_high`, `trace_id`) COMMENT
'for getTraces/ByIds';
ALTER TABLE zipkin_annotations ADD INDEX(`endpoint_service_name`) COMMENT 'for
getTraces and getServiceNames';
ALTER TABLE zipkin_annotations ADD INDEX(`a_type`) COMMENT 'for getTraces';
ALTER TABLE zipkin_annotations ADD INDEX(`a_key`) COMMENT 'for getTraces';
ALTER TABLE zipkin_annotations ADD INDEX(`trace_id`, `span_id`, `a_key`) COMMENT
'for dependencies job';

CREATE TABLE IF NOT EXISTS zipkin_dependencies (
`day` DATE NOT NULL,
`parent` VARCHAR(255) NOT NULL,
`child` VARCHAR(255) NOT NULL,
`call_count` BIGINT
) ENGINE=InnoDB ROW_FORMAT=COMPRESSED CHARACTER SET=utf8 COLLATE utf8_general_ci;

ALTER TABLE zipkin_dependencies ADD UNIQUE KEY(`day`, `parent`, `child`);

```

创建好表之后，使用 Navicat 检查在 Zipkin 数据库中是否生成了三张表，如图 12-11 所示。

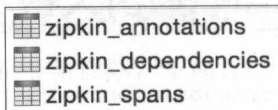


图12-11 Zipkin数据库表

创建好表之后，我们要去改动一下 4 个服务的代码。

12.4.2 使用 Brave-MySQL 存储追踪信息

分别对 4 个服务做以下操作。

在 pom.xml 中添加 Brave-MySQL 依赖，该依赖是 Brave 提供的，代码如下：

```
<!-- zipkin-mysql -->
<dependency>
  <groupId>io.zipkin.brave</groupId>
  <artifactId>brave-mysql</artifactId>
  <version>3.9.0</version>
</dependency>
```

在 ZipkinConfig 类中添加 MySQLStatementInterceptorManagementBean，代码如下：

```
@Bean
public MySQLStatementInterceptorManagementBean
mysqlStatementInterceptorManagementBean(Brave brave) {
    return new MySQLStatementInterceptorManagementBean(brave.clientTracer());
}
```

代码编写完成之后，使用以下命令启动 Zipkin：

```
STORAGE_TYPE=mysql MYSQL_HOST=192.192.19.192 MYSQL_TCP_PORT=3306
MYSQL_DB=zipkin MYSQL_USER=root MYSQL_PASS=123456 nohup java -jar
/opt/zipkin-server-1.5.1-exec.jar &
```

其中，STORAGE_TYPE 指定使用 MySQL 来存储；MYSQL_HOST 和 MYSQL_TCP_PORT 指定 MySQL 的地址；MYSQL_DB 指定使用哪一个数据库；MYSQL_USER 和 MYSQL_PASS 指定了 MySQL 的用户名和密码；最后启动服务。

之后，启动 4 个服务，通过 Swagger 进行测试，观察数据库表中记录的变化。

这里，使用了 4 个服务。访问一次 myservice1，在 zipkin_annotations 表中会增加 25 条记录，这是一件恐怖的事。在高并发情况下，这张表很快就会被打爆。所以在生产环境中，可以使用 Cassandra 进行存储。

12.5 使用 Brave-OkHttp 实现全链路追踪

接下来，介绍在使用 OkHttp 进行通信的情况下，怎么使用 Zipkin。为了简单起见，新

建一个服务 myservice5。myservice5 调用 12.3 节中的 myservice4 返回给用户信息。

12.5.1 搭建项目框架

myservice5 的代码结构与 myservice4 相似，不再赘述。直接来看 pom.xml 文件，内容如下：

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.
w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.
apache.org/maven-v4_0_0.xsd">
    <modelVersion>4.0.0</modelVersion>

    <groupId>com.microservice</groupId>
    <artifactId>myservice5</artifactId>
    <version>1.0-SNAPSHOT</version>

    <properties>
        <java.version>1.8</java.version>
    </properties>
    <parent>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-parent</artifactId>
        <version>1.4.3.RELEASE</version>
    </parent>
    <dependencies>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-web</artifactId>
        </dependency>
        <dependency>
            <groupId>io.springfox</groupId>
            <artifactId>springfox-swagger2</artifactId>
            <version>2.2.2</version>
        </dependency>
        <dependency>
            <groupId>io.springfox</groupId>
```

```

        <artifactId>springfox-swagger-ui</artifactId>
        <version>2.2.2</version>
    </dependency>
    <dependency>
        <groupId>io.zipkin.brave</groupId>
        <artifactId>brave-core</artifactId>
        <version>3.9.0</version>
    </dependency>
    <dependency>
        <groupId>io.zipkin.brave</groupId>
        <artifactId>brave-spancollector-http</artifactId>
        <version>3.9.0</version>
    </dependency>
    <dependency>
        <groupId>io.zipkin.brave</groupId>
        <artifactId>brave-web-servlet-filter</artifactId>
        <version>3.9.0</version>
    </dependency>
    <dependency>
        <groupId>io.zipkin.brave</groupId>
        <artifactId>brave-okhttp</artifactId>
        <version>3.9.0</version>
    </dependency>
</dependencies>

<build>
    <plugins>
        <plugin>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-maven-plugin</artifactId>
        </plugin>
    </plugins>
</build>
</project>

```

除了引入 `spring-boot-starter-web`、`Swagger` 及 `Zipkin` 的三个依赖之外，还引入了 `Brave-OkHttp` 依赖，该依赖也是 `Brave` 提供的，是一个在 `OkHttpClient` 中集成了 `Brave` 的依赖。

引入依赖之后，创建服务启动主类，代码如下：

```
package com.microservice.myservice5;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import springfox.documentation.swagger2.annotations.EnableSwagger2;

@SpringBootApplication
@EnableSwagger2
public class Application {
    public static void main(String[] args) {
        SpringApplication sa = new SpringApplication(Application.class);
        sa.run(args);
    }
}
```

12.5.2 使用服务端与客户端拦截器收集追踪信息

使用 Brave-OkHttp 后，客户端拦截器的添加将变得很简单，代码如下：

```
package com.microservice.myservice5.config;

import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;

import com.github.kristofa.brave.Brave;
import com.github.kristofa.brave.EmptySpanCollectorMetricsHandler;
import com.github.kristofa.brave.Sampler;
import com.github.kristofa.brave.SpanCollector;
import com.github.kristofa.brave.http.DefaultSpanNameProvider;
import com.github.kristofa.brave.http.HttpSpanCollector;
import com.github.kristofa.brave.okhttp.BraveOkHttpRequestResponseInterceptor;
import com.github.kristofa.brave.servlet.BraveServletFilter;

import okhttp3.OkHttpClient;

@Configuration
public class ZipkinConfig {
```

```

@Bean
public SpanCollector spanCollector() {
    HttpSpanCollector.Config spanConfig =
        HttpSpanCollector.Config.builder()
            .compressionEnabled(false)
            .connectTimeout(5000)
            .flushInterval(1)
            .readTimeout(6000).build();

    return HttpSpanCollector.create("http://10.211.55.13:9411", spanConfig,
        new EmptySpanCollectorMetricsHandler());
}

@Bean
public Brave brave(SpanCollector spanCollector) {
    Brave.Builder builder = new Brave.Builder("myservice5");
    builder.spanCollector(spanCollector);
    builder.traceSampler(Sampler.create(1));
    return builder.build();
}

@Bean
public BraveServletFilter braveServletFilter(Brave brave) {
    return new BraveServletFilter(brave.serverRequestInterceptor(),
        brave.serverResponseInterceptor(),
        new DefaultSpanNameProvider());
}

@Bean
public OkHttpClient okHttpClient(Brave brave) {
    return new OkHttpClient.Builder()
        .addInterceptor(new
        BraveOkHttpRequestResponseInterceptor(brave.clientRequestInterceptor(),
            brave.clientResponseInterceptor(), new
            DefaultSpanNameProvider()))
        .build();
}
}

```

除了创建了 `spanCollector`、`brave` 和 `braveServletFilter` 3 个 Bean 之外，还创建了一个

okHttpClient 的 Bean, 在该 Bean 中添加了 clientRequestInterceptor 和 clientResponseInterceptor。在使用 AsyncHttpClient 通信时, 对于这两个拦截器需要自己写一些代码, 但是使用 OkHttp, 一切都有现成的。

最后, 编写 controller 验证链路追踪功能。代码如下:

```
package com.microservice.myservice5.controller;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestMethod;
import org.springframework.web.bind.annotation.RestController;

import io.swagger.annotations.Api;
import okhttp3.OkHttpClient;
import okhttp3.Request;
import okhttp3.Response;

import java.io.IOException;

@Api("zipkin-brave 相关 api")
@RestController
@RequestMapping("/myservice5/zipkin")
public class Myservice5Controller {
    @Autowired
    private OkHttpClient okHttpClient;

    @RequestMapping(value = "/test", method = RequestMethod.GET)
    public String test() {
        Response response = null;
        try {
            Request request = new Request.Builder().url("http://localhost:8084/
myservice4/zipkin/test").build();
            response = okHttpClient.newCall(request).execute();
            return response.body().string();
        } catch (IOException e) {
            e.printStackTrace();
        } finally {
            if (response != null && response.body() != null) {
                response.body().close();
            }
        }
    }
}
```



```

    }
    }
    return "";
}
}

```

依然是熟悉的调用方式，在该 controller 中看不到一点儿 Zipkin 的影子。之后，使用 Swagger 进行测试就可以了，这里给出一张测试后的依赖图，如图 12-12 所示。

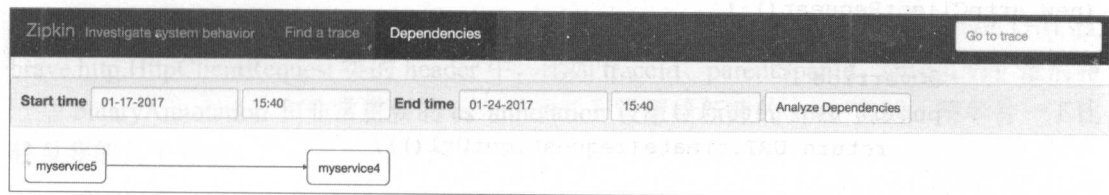


图12-12 Zipkin服务依赖图

12.6 再学一招：Brave 关键源码解析

Brave 的关键点有三个：使用 reporter 创建 span；使用 collector 收集这些 span；使用 collector 将收集到的 span 发送给 Zipkin-server。

说白了，链路追踪的核心就是 span。一条完整的链路就是多个 span 组合起来的几次调用。首先来看一下 span 的生命周期。

12.6.1 span 的生命周期

一个 span 的生命周期如图 12-13 所示。

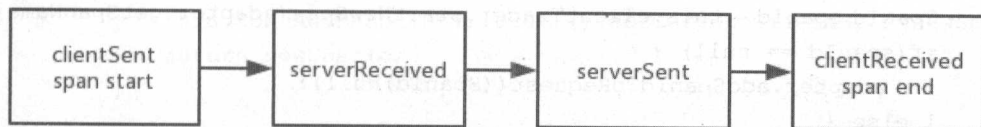


图12-13 span生命周期图

从 cs 开始创建一个 span 到 cr 并 reporter 一个 span 给 collector，最后消除 span。以上 4 个过程对应于 Brave 的 4 个拦截器 clientRequestInterceptor、ServerRequestInterceptor、ServerResponseInterceptor 及 clientResponseInterceptor。

首先，来看一下 span 的创建。

12.6.2 使用 reporter 创建 span

span 的创建是在 `clientRequestInterceptor` 拦截器中完成的，该拦截器的使用姿势如下：

```
private void clientRequestInterceptor(Request request) {
    brave.clientRequestInterceptor().handle(new HttpClientRequestAdapter
(new HttpClientRequest() {

        @Override
        public URI getUri() {
            return URI.create(request.getUrl());
        }

        @Override
        public String getHttpMethod() {
            return request.getMethod();
        }

        @Override
        public void addHeader(String headerKey, String headerValue) {
            request.getHeaders().add(headerKey, headerValue);
        }

    }, new DefaultSpanNameProvider()));
}
```

直接来看其最重要的方法 `handle(ClientRequestAdapter adapter)`：

```
public void handle(ClientRequestAdapter adapter) {
    SpanId spanId = this.clientTracer.startNewSpan(adapter.getSpanName());
    if (spanId == null) {
        adapter.addSpanIdToRequest((SpanId) null);
    } else {
        adapter.addSpanIdToRequest(spanId);
        Iterator var3 = adapter.requestAnnotations().iterator();
        while (var3.hasNext()) {
            KeyValueAnnotation annotation = (KeyValueAnnotation) var3.next();
            this.clientTracer.submitBinaryAnnotation(annotation.getKey(),
```

```

annotation.getValue());
    }

    this.recordClientSentAnnotations(adapter.serverAddress());
}

}

```

在该方法中,首先新建了一个 span 实例,为该 span 设置 spanId、parentSpanId 和 traceId,名字为上边的 getHttpMethod()方法的返回值;之后将一系列的数据加入 com.github.kristofa.brave.http.HttpClientRequest 类的 header 中,比如 traceId、parentSpanId、spanId 等;最后将一些 binaryAnnotation 和非常重要的 cs annotation 设置给新建的 span 实例。简单看一下比较重要的几个方法。

第一个方法创建新的 span:

```

public SpanId startNewSpan(String requestName) {
    Boolean sample = this.spanAndEndpoint().state().sample();
    if(Boolean.FALSE.equals(sample)) {
        this.spanAndEndpoint().state().setCurrentClientSpan((Span)null);
        return null;
    } else {
        SpanId newSpanId = this.getNewSpanId();
        if(sample == null
        && !this.traceSampler().isSampled(newSpanId.traceId)) {
            this.spanAndEndpoint().state().setCurrentClientSpan((Span)null);
            return null;
        } else {
            Span newSpan = newSpanId.toSpan();
            newSpan.setName(requestName);
            this.spanAndEndpoint().state().setCurrentClientSpan(newSpan);
            return newSpanId;
        }
    }
}

private SpanId getNewSpanId() {
    Span parentSpan = this.spanAndEndpoint().state().getCurrentLocalSpan();
    if(parentSpan == null) {
        ServerSpan newSpanId =

```

```

this.spanAndEndpoint().state().getCurrentServerSpan();
    if(newSpanId != null) {
        parentSpan = newSpanId.getSpan();
    }

    long newSpanId1 = this.randomGenerator().nextLong();
    com.github.kristofa.brave.SpanId.Builder builder = SpanId.builder().
    spanId(newSpanId1);
    return parentSpan == null?builder.build():builder.traceId(parentSpan.
    getTrace_id()).parentId(Long.valueOf(parentSpan.getId())).build();
}

```

该方法的实现比较曲折,先创建一个 `SpanId` 实例,并且为该 `SpanId` 实例初始化了 `traceId` 和 `parentSpanId`; 之后使用 `newSpanId.toSpan()` 创建一个 `Span` 实例,最后将之前的 `SpanId` 实例中的 `traceId`、`parentId` 及 `spanId` 初始化给 `span`。如下:

```

public Span toSpan() {
    Span result = new Span();
    result.setId(this.spanId);
    result.setTrace_id(this.traceId);
    result.setParent_id(this.nullableParentId());
    if(this.debug()) {
        result.setDebug(Boolean.valueOf(this.debug()));
    }

    return result;
}

```

这样一个 `span` 就创建完成了!

第二个比较重要的方法如下:

```

public void addSpanIdToRequest(@Nullable SpanId spanId) {
    if(spanId == null) {
        this.request.addHeader(BraveHttpHeaders.Sampled.getName(), "0");
    } else {
        this.request.addHeader(BraveHttpHeaders.Sampled.getName(), "1");
        this.request.addHeader(BraveHttpHeaders.TraceId.getName(),
        IdConversion.convertToString(spanId.traceId));
    }
}

```

```

        this.request.addHeader(BraveHttpHeaders.SpanId.getName(),
IdConversion.convertToString(spanId.spanId));
        if(spanId.nullableParentId() != null) {
            this.request.addHeader(BraveHttpHeaders.ParentSpanId.getName(),
IdConversion.convertToString(spanId.parentId));
        }
    }
}

```

这里先给出 `addSpanIdToRequest` 方法中用到的枚举：

```

public enum BraveHttpHeaders {
    TraceId("X-B3-TraceId"),
    SpanId("X-B3-SpanId"),
    ParentSpanId("X-B3-ParentSpanId"),
    Sampled("X-B3-Sampled");

    private final String name;

    private BraveHttpHeaders(String name) {
        this.name = name;
    }

    public String getName() {
        return this.name;
    }
}

```

`addSpanIdToRequest` 方法在 `HttpClientRequest` 类的 `header` 中，添加了“X-B3-TraceId”、“X-B3-SpanId”、“X-B3-ParentSpanId”以及“X-B3-Sampled”。

第三个比较重要的方法是 `recordClientSentAnnotations`：

```

private void recordClientSentAnnotations(Endpoint serverAddress) {
    if(serverAddress == null) {
        this.clientTracer.setClientSent();
    } else {
        this.clientTracer.setClientSent(serverAddress.ipv4,
serverAddress.port.shortValue(), serverAddress.service_name);
    }
}

```



```
}

```

这里调用了 `ClientTracer` 的 `setClientSent` 方法，如下：

```
public void setClientSent() {
    this.submitStartAnnotation("cs");
}

void submitStartAnnotation(String annotationName) {
    Span span = this.spanAndEndpoint().span();
    if(span != null) {
        Annotation annotation = Annotation.create(this.currentTimeMicroseconds(),
annotationName, this.spanAndEndpoint().endpoint());
        synchronized(span) {
            span.setTimestamp(Long.valueOf(annotation.timestamp));
            span.addToAnnotations(annotation);
        }
    }
}
```

该方法比较简单，组建 `annotation`，并且为 `annotation` 初始化参数，比如 `timestamp`、`value` 及 `endpoint`，其中，`value` 就是 `cs`，最后将该 `annotation` 添加到新建的 `span` 实例的 `List<Annotation>` 中。

至此，`clientRequestInterceptor` 的源码就解析完了。接下来看一下 `ServerRequestInterceptor` 和 `ServerResponseInterceptor` 的源码。

由于篇幅原因，这里只列出 `BraveServletFilter` 在 `doFilter` 方法中的使用，内容比较简单，具体源码不再详述了。

```
public void doFilter(ServletRequest request, ServletResponse response,
FilterChain filterChain) throws IOException, ServletException {
    String alreadyFilteredAttributeName =
this.getAlreadyFilteredAttributeName();
    boolean hasAlreadyFilteredAttribute =
request.getAttribute(alreadyFilteredAttributeName) != null;
    if(hasAlreadyFilteredAttribute) {
        filterChain.doFilter(request, response);
    }
}
```

```

    } else {
        final BraveServletFilter.StatusExposingServletResponse
statusExposingServletResponse = new BraveServletFilter.StatusExposingServlet-
Response((HttpServletResponse) response);
        this.requestInterceptor.handle(new HttpServerRequestAdapter(new
ServletHttpRequest((HttpServletRequest) request), this.spanNameProvider));

        try {
            filterChain.doFilter(request, statusExposingServletResponse);
        } finally {
            this.responseInterceptor.handle(new HttpServerResponseAdapter
(new HttpServletResponse() {
                public int getHttpStatusCode() {
                    return statusExposingServletResponse.getStatus();
                }
            }));
        }
    }
}

```

12.6.3 使用 collector 收集 span

collector 收集 span 的动作是在 `clientResponseInterceptor` 中完成的, 首先来看一下 `clientResponseInterceptor` 的正确使用方法:

```

private void clientResponseInterceptor(Response response) {
    brave.clientResponseInterceptor().handle(new HttpClientResponseAdapter
(new HttpServletResponse() {
        public int getHttpStatusCode() {
            return response.getStatusCode();
        }
    }));
}

```

看一下 `handle` 方法:

```

public void handle(ClientResponseAdapter adapter) {
    try {
        Iterator var2 = adapter.responseAnnotations().iterator();
    }
}

```

```

        while(var2.hasNext()) {
            KeyValueAnnotation annotation = (KeyValueAnnotation)var2.next();
            this.clientTracer.submitBinaryAnnotation(annotation.getKey(),
annotation.getValue());
        }
    } finally {
        this.clientTracer.setClientReceived();
    }
}

```

该方法做的最主要的事情是调用 `ClientTracer` 的 `setClientReceived` 方法，源码如下：

```

public void setClientReceived() {
    if(this.submitEndAnnotation("cr", this.spanCollector())) {
        this.spanAndEndpoint().state().setCurrentClientSpan((Span)null);
    }
}

boolean submitEndAnnotation(String annotationName, SpanCollector
spanCollector) {
    Span span = this.spanAndEndpoint().span();
    if(span == null) {
        return false;
    } else {
        Annotation annotation = Annotation.create(this.currentTimeMicroseconds(),
annotationName, this.spanAndEndpoint().endpoint());
        span.addToAnnotations(annotation);
        if(span.getTimestamp() != null) {
            span.setDuration(Long.valueOf(annotation.timestamp -
span.getTimestamp().longValue()));
        }

        spanCollector.collect(span);
        return true;
    }
}

```

这里做了两件重要的事情，第一件事是创建一个 value 是 `cr` 的 `annotation`，第二件事是使用 `collector` 收集数据。这里我们使用了 `FlushingSpanCollector` 的 `collect` 方法：

```
private final BlockingQueue<Span> pending = new LinkedBlockingQueue(1000);

public void collect(Span span) {
    this.metrics.incrementAcceptedSpans(1);
    if(!this.pending.offer(span)) {
        this.metrics.incrementDroppedSpans(1);
    }
}
}
```

逻辑很简单，将 span 放到 pending 队列中，然后 collector 会从该队列中取出数据发送给 Zipkin-server。

事实上，collector 在两个点收集数据，一个点是在 `ServerResponseInterceptor` 的 `handle` 方法中，另一个点就是在 `clientResponseInterceptor` 的 `handle` 方法中。

至此，reporter 创建 span 并调用 collector 收集 span 的源码就解析完了。

12.6.4 使用 collector 发送 span

在 `ZipkinConfig` 类中，在创建 `HttpSpanCollector` 实例的时候，调用了如下方法：

```
HttpSpanCollector create(String baseUrl, Config config,
    SpanCollectorMetricsHandler metrics)
```

简单看一下与该方法相关的关键源码，该方法调用了 `HttpSpanCollector` 的父类的 `FlushingSpanCollector` 构造器：

```
protected FlushingSpanCollector(SpanCollectorMetricsHandler metrics, int
flushInterval) {
    this.metrics = metrics;
    this.flusher = flushInterval > 0 ? new FlushingSpanCollector.Flusher(this,
flushInterval) : null;
}
```

在该构造器中，当设置的 `flushInterval > 0` 时，就开始执行 `Flusher` 构造器，`Flush` 是 `FlushingSpanCollector` 的一个静态内部类，其实现了 `Runnable` 接口，内部类如下：

```
static final class Flusher implements Runnable {
    final Flushable flushable;
```



```

final ScheduledExecutorService scheduler = Executors.newScheduledThreadPool(1);

Flusher(Flushable flushable, int flushInterval) {
    this.flushable = flushable;
    this.scheduler.scheduleWithFixedDelay(this, 0L, (long)flushInterval,
TimeUnit.SECONDS);
}

public void run() {
    try {
        this.flushable.flush();
    } catch (IOException var2) {}
}
}

```

在 `Flusher` 的构造器中,启动了一个定时任务,每隔 `flushInterval` 指定的时间执行一次。具体执行的逻辑在 `FlushingSpanCollector` 的 `flush()`方法中,看一下该方法:

```

private final BlockingQueue<Span> pending = new LinkedBlockingQueue(1000);

public void flush() {
    if(!this.pending.isEmpty()) {
        ArrayList drained = new ArrayList(this.pending.size());
        this.pending.drainTo(drained);
        if(!drained.isEmpty()) {
            int spanCount = drained.size();

            try {
                this.reportSpans(drained);
            } catch (IOException var4) {
                this.metrics.incrementDroppedSpans(spanCount);
            } catch (RuntimeException var5) {
                this.metrics.incrementDroppedSpans(spanCount);
            }
        }
    }
}

```



```

    }
    }
}

```

在该方法中，首先将 **pending** 队列中的 **span** 全部存到 **drained** 集合中，之后调用其子类 **AbstractSpanCollector** 的 **reportSpans** 方法，代码如下：

```

protected void reportSpans(List<Span> drained) throws IOException {
    byte[] encoded = this.codec.writeSpans(drained);
    this.sendSpans(encoded);
}

```

在该方法中，首先将 **span** 集中的 **span** 转换成 Zipkin 的通用 **span**，之后进行序列化，最后调用其子类 **HttpSpanCollector** 的 **sendSpans** 方法，将 **span** 信息真正地发送给 Zipkin-server。

```

@Override
protected void sendSpans(byte[] json) throws IOException {
    HttpURLConnection connection = (HttpURLConnection) new URL(url).openConnection();
    connection.setConnectTimeout(config.connectTimeout());
    connection.setReadTimeout(config.readTimeout());
    connection.setRequestMethod("POST");
    connection.addRequestProperty("Content-Type", "application/json");
    if (config.compressionEnabled()) {
        connection.addRequestProperty("Content-Encoding", "gzip");
        ByteArrayOutputStream gzipped = new ByteArrayOutputStream();
        try (GZIPOutputStream compressor = new GZIPOutputStream(gzipped)) {
            compressor.write(json);
        }
        json = gzipped.toByteArray();
    }
    connection.setDoOutput(true);
    connection.setFixedLengthStreamingMode(json.length);
    connection.getOutputStream().write(json);

    try (InputStream in = connection.getInputStream()) {
        while (in.read() != -1) ; // skip
    } catch (IOException e) {

```

```
try (InputStream err = connection.getErrorStream()) {  
    if (err != null) { // possible, if the connection was dropped  
        while (err.read() != -1) ; // skip  
    }  
    throw e;  
}
```

该方法使用了一个长连接。至此，collector 向 Zipkin 发送 span 的源码就解析完了。

第 13 章

微服务持续集成与持续部署系统

13.1 初识持续集成与持续部署系统

这里所说的持续集成与持续部署系统范围比较广，其功能包括代码管理、版本控制、自动编译打包及自动部署等，当然，如果服务按照 Docker 镜像来部署，则还包括自动打包成镜像文件以及自动 push 到镜像仓库等步骤。

不管是在微服务架构中，还是在传统的单体架构中，搭建一套持续集成与持续部署系统都是必要的，因为这会大大减少人力工作。如果没有持续集成与持续部署系统，我们需要自己管理代码，进行版本控制，如果一个服务的版本很多，或者代码量很大，就变得非常不好管理；如果没有持续集成与持续部署系统，我们需要手动将代码打成 jar 包，手动上传到一台服务器，手动关掉之前的服务，手动启动 jar 包进程，如果按照 Docker 镜像来部署，我们可能还需要手动将服务打包成镜像，手动将镜像文件 push 到镜像仓库，手动将镜像文件从镜像仓库 pull 下来，手动将镜像运行起来。这一切都是手动的！效率极低，并且容易出错。所以，搭建一套持续集成与持续部署系统是非常有必要的！下面我们就开启本章的持续集成与持续部署之旅！

13.2 系统总体架构

首先来看一下本章将要搭建的持续集成与持续部署系统的总体架构，如图 13-1 所示。

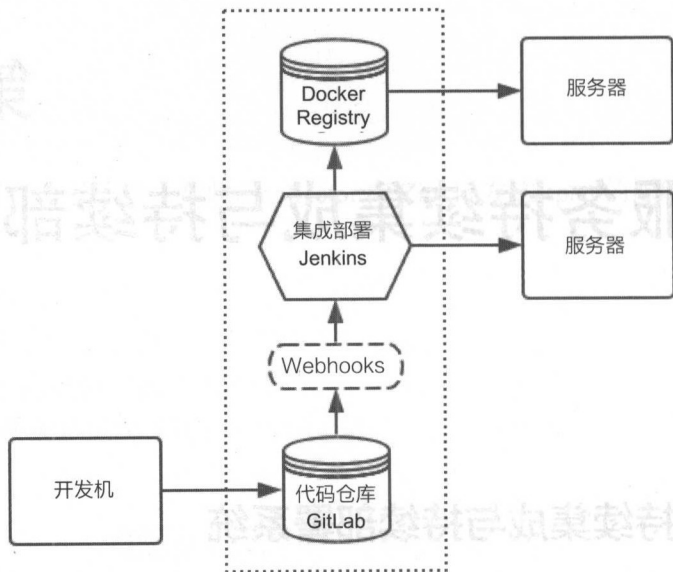


图13-1 持续集成与持续部署系统的总体架构

在整个架构中，主要包括三大组件：GitLab、Jenkins 及 Docker-Registry。还有一个 webhooks，这是用来轻松连接 GitLab 和 Jenkins 的纽带，它可以非常方便地实现持续集成和持续部署。如果不使用 Docker 部署，则不需要安装搭建 Docker-Registry。下面依次介绍各个组件。

13.2.1 初识 GitLab

GitLab 是一款主要用于代码管理的工具，是目前为止最受企业欢迎的代码管理工具。GitLab 包含 Git 仓库管理、问题追踪、活动流、代码重审及 wiki 等功能。相较于 GitHub，可以自己搭建服务器，这可以避免因为网络速度慢导致部署效率低下，同时，自己搭建服务器，安全性更高。GitLab 也内建了持续集成与持续部署的功能，例如，非常好用的 webhook，它使得与 Jenkins 的集成非常简单。

13.2.2 初识 Jenkins

Jenkins 主要用于管理版本，进行代码的编译和部署，使用 webhook 插件可以实现与 GitLab 的集成，实现持续集成与持续部署。Jenkins 有成百上千的插件可以使用，可扩展性极强；Jenkins 的安装也很简单，除了使用 Docker 镜像之外，还可以直接使用 war 包部署；并且提供了 webUI，使我们可以方便地进行配置信息的管理；Jenkins 可以实现分布式部署，这在服务数量比较多的时候很有用，可以分散压力到各个 Jenkins 机器上，从而减轻 Jenkins 的各个机器的压力，也使得服务的编译和部署的速度变快（因为不需要排队）。

13.2.3 初识 Docker-Registry

Docker-Registry 是存放 Docker 镜像的仓库，可以类比 GitLab。Docker-Registry 相较于 DockerHub 可以类比 GitLab 相较于 GitHub，主要是在这里可以自己搭建镜像仓库，安全性更高，网络速度更快。如果你向 DockerHub 推过镜像就知道，速度通常慢得令人发指。Docker-Registry 提供了便捷的 API，可以让我们方便地进行镜像及版本的查询。

13.3 持续集成与持续部署系统工作原理

介绍完各个组件之后，来看一下整套持续集成与持续部署系统的工作原理，假设我们已经配置好了各个组件。

13.3.1 使用 jar 包部署项目的整体流程

第一步，开发人员使用 Git 客户端，将代码 push 到 GitLab。

第二步，GitLab 的 webhook 插件会通知 Jenkins 进行工作。

第三步，Jenkins 从 GitLab 上拉取代码并使用 Maven 进行编译打包。

第四步，Jenkins 将 jar 包发到服务器。

第五步，Jenkins 调用服务器上的 Shell 脚本停止之前的服务，并启动 jar 进程。

13.3.2 使用 Docker 镜像部署项目的整体流程

第一步，开发人员使用 Git 客户端，将代码 push 到 GitLab。

第二步，GitLab 的 webhook 插件会通知 Jenkins 进行工作。

第三步, Jenkins 从 GitLab 上拉取代码并使用 docker-maven-plugin 进行编译打包, 最后将 jar 包打成镜像。

第四步, Jenkins 将镜像 push 到 Docker-Registry。

第五步, Jenkins 调用服务器上的 Shell 脚本从 Docker-Registry 拉取镜像, 然后停止之前的服务, 最后启动 Docker 镜像。

13.4 搭建持续集成与持续部署系统

安装环境及软件版本

- 操作系统: centos7, ip 是 10.211.55.4
- Docker: 1.12.3
- GitLab: 8.13.1
- Jenkins: 2.19.1
- Docker-Registry: 2.5.0

13.4.1 安装启动 Docker

第一步, 添加 yum 源。

```
tee /etc/yum.repos.d/docker.repo <<-'EOF'
[dockerrepo]
name=Docker Repository
baseurl=https://yum.dockerproject.org/repo/main/centos/7/
enabled=1
gpgcheck=1
gpgkey=https://yum.dockerproject.org/gpg
EOF
```

添加 yum 源之后, 可以在/etc/yum.repos.d/中找到该 yum 源。

第二步, 安装 Docker。

```
yum install docker-engine
```

第三步, 设置 Docker 开机启动。

```
systemctl enable docker.service
```

第四步，启动 Docker。

```
systemctl start docker
```

13.4.2 安装配置启动 GitLab

手工搭建 GitLab 比较费劲，这里直接使用 Docker 镜像进行安装。首先下载 GitLab 镜像，命令如下：

```
docker pull gitlab/gitlab-ce
```

这个过程需要一段时间，下载之后，使用如下命令查看下载的镜像信息。不熟悉 Docker 命令无所谓，在本章的“再学一招”部分，笔者会把常用的 Docker 命令列出来。

```
docker images
```

之后看到如图 13-2 所示的信息。

[root@centos-linux-2 ~]# docker images				
REPOSITORY	TAG	IMAGE ID	CREATED	SIZE
gitlab/gitlab-ce	latest	e5b990ac5dff	3 months ago	1.22 GB

图13-2 Docker镜像文件列表

其中，REPOSITORY 指定了镜像来源于哪个仓库；TAG 指定了镜像的版本（latest 表明该版本是最新的版本）；IMAGE ID 指定了镜像的标识 ID。

之后使用如下命令预启动 GitLab：

```
docker run -d -h gitlab.zhaojigang.com -p 80:80 -v /etc/gitlab:/etc/gitlab/ -v /var/log/gitlab:/var/log/gitlab/ -v /var/opt/gitlab:/var/opt/gitlab/ --name gitlab gitlab/gitlab-ce
```

各参数的意义如下。

- -d：该镜像以后台容器运行。
- -h gitlab.zhaojigang.com：浏览器通过该 host 访问 GitLab。
- -p 80:80：浏览器通过该 port 访问 GitLab。

- `-v /etc/gitlab:/etc/gitlab/`: 将 GitLab 容器内部的 `/etc/gitlab/` 目录挂载到本机的 `/etc/gitlab/` 目录。操作本机的 `/etc/gitlab/` 目录相当于操作 GitLab 容器内部的 `/etc/gitlab/` 目录。
- `--name gitlab`: 启动的容器名称为 GitLab。
- 最后的 `gitlab/gitlab-ce` 表示基于该镜像进行容器创建。

本次启动称为预启动, 因为本次启动是为了在本机生成如下 3 个配置文件。

- `/etc/gitlab/`: 配置文件所在的目录。
- `/var/log/gitlab/`: 日志所在目录。
- `/var/opt/gitlab/`: 数据所在目录。

这 3 个配置文件创建好之后, 开始配置 GitLab。修改 `/etc/gitlab/gitlab.rb` 文件, 其中 `external_url` 修改如下:

```
external_url 'http://gitlab.zhaojigang.com:8929'
```

该配置指定了外部浏览器访问 GitLab 的 url。这里将 http 端口设置为 8929 而不是默认的 80。

之后, 删除之前的 GitLab 容器 (删除命令在本章的“再学一招”部分介绍), 再使用如下命令启动 GitLab:

```
docker run -d -h gitlab.zhaojigang.com -p 8929:8929 -v /etc/gitlab:/etc/gitlab/
-v /var/log/gitlab:/var/log/gitlab/ -v /var/opt/gitlab:/var/opt/gitlab/
--name gitlab gitlab/gitlab-ce
```

启动端口是 8929。启动完成后, 使用如下命令查看 GitLab 容器是否已经被创建并运行:

```
docker ps
```

看到如图 13-3 所示的信息。

CONTAINER ID	IMAGE	COMMAND	CREATED	STATUS	PORTS	NAMES
6cbd10274198	gitlab/gitlab-ce	"/assets/wrapper"	5 seconds ago	Up 4 seconds	22/tcp, 80/tcp, 443/tcp, 0.0.0.0:8929->8929/tcp	gitlab

图13-3 Docker容器信息列表

这表示 GitLab 容器被正常创建并运行。其中容器 ID 为 6cbd10274198, 容器 name 为 GitLab。这两个参数是我们操作容器的必要参数。

启动完成之后，可以通过在浏览器中输入“<http://gitlab.zhaojigang.com:8929/>”来访问 GitLab。在访问之前，首先需要在开发机上处理一下域名和 ip 的映射关系。在 `/etc/hosts` 文件中添加如下内容：

```
10.211.55.4    gitlab.zhaojigang.com
```

之后，使用浏览器访问，在 GitLab 登录页注册登录后，进入 GitLab 主页，如图 13-4 所示。

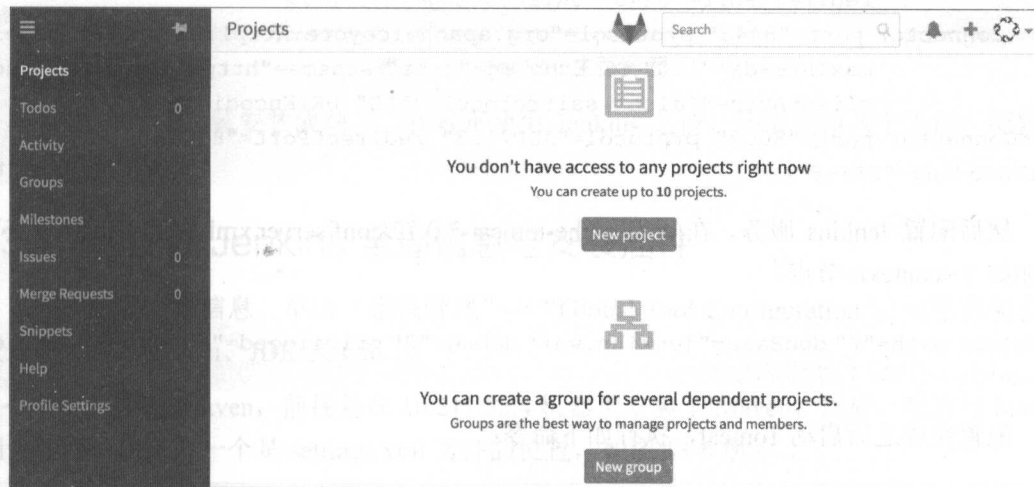


图13-4 GitLab初始页面

至此，GitLab 就安装成功了！下面安装 Jenkins。

13.4.3 安装启动 Jenkins

Jenkins 的安装很简单，通常有两种方式：使用 Docker 镜像安装和使用 war 包安装。如果在只使用 jar 包部署项目的情况下，二者都没有问题。但是如果使用 Docker 镜像部署项目，Jenkins 如果又使用 Docker 镜像安装就需要使用到 Docker-in-Docker 技术，该技术并没有经过太多的考验，所以，笔者这里使用 war 包部署 Jenkins。

首先，需要在 10.211.55.4 上安装 Tomcat。假设安装后的目录是 `/opt/apache-tomcat-7.0.72`。之后从 Jenkins 官网 (<https://jenkins.io/>) 下载 Jenkins 的 war 包。下载之后，将该 war 包放在 `/opt/apache-tomcat-7.0.72/webapps` 文件夹下。最后，配置 Tomcat 文件。首先需要配置 Tomcat 编码。在 `/opt/apache-tomcat-7.0.72/conf/server.xml` 中的 `<connector>` 节点下添加

URIEncoding="UTF-8"。如下：

```
<Connector port="8080" protocol="HTTP/1.1"
    connectionTimeout="20000"
    redirectPort="8443" URIEncoding="UTF-8"/>
<Connector executor="tomcatThreadPool"
    port="8080" protocol="HTTP/1.1"
    connectionTimeout="20000"
    redirectPort="8443" URIEncoding="UTF-8"/>
<Connector port="8443" protocol="org.apache.coyote.http11.Http11Protocol"
    maxThreads="150" SSLEnabled="true" scheme="https" secure="true"
    clientAuth="false" sslProtocol="TLS" URIEncoding="UTF-8"/>
<Connector port="8009" protocol="AJP/1.3" redirectPort="8443"
URIEncoding="UTF-8"/>
```

然后配置 Jenkins 服务。在/opt/apache-tomcat-7.0.72/conf/server.xml 中的<Host>节点下添加如下<context>节点：

```
<context path="/" docBase="jenkins.war" debug="0" privileged="true" reloadable=
"true"/>
```

配置完成之后启动 Tomcat，执行如下命令：

```
cd /opt/apache-tomcat-7.0.72/bin
nohup ./startup.sh &
```

启动 Tomcat 之后，在浏览器中输入“http://10.211.55.4:8080/jenkins/”进行访问，就可以看到 Jenkins 的启动页面，如果无法访问，关闭防火墙。

```
systemctl disable firewalld
systemctl stop firewalld
```

之后还需要解锁 Jenkins。输入/root/.jenkins/secrets/initialAdminPassword 中的密码来解锁 Jenkins。其中，/root/.jenkins 是 Jenkins 默认的主目录。

注意，在 Jenkins 的启动过程中，会提醒我们在线下载各种插件，除非你的网速很快，否则不要在线安装 Jenkins 插件。

Jenkins 启动之后，创建用户并登录。登录成功之后，会进入到 Jenkins 首页，如图 13-5 所示。

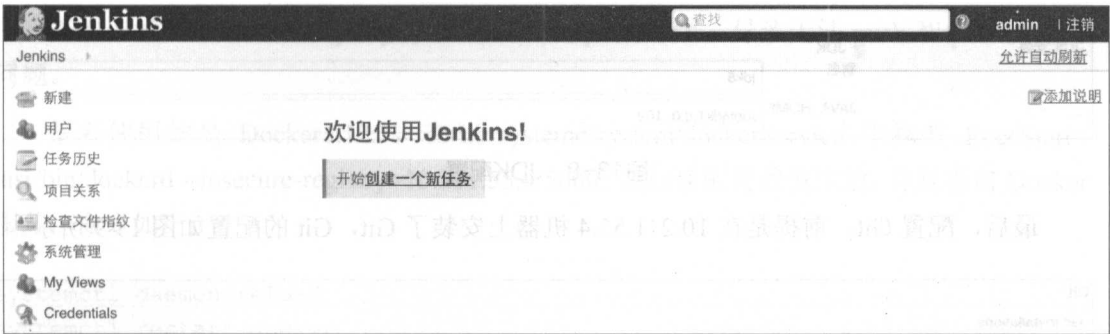


图13-5 Jenkins主页界面

至此，Jenkins 就安装成功了。在真正使用 Jenkins 之前，还需要做两个工作：配置全局信息和安装插件。

13.4.4 配置 Jenkins 全局信息与安装插件

先来配置全局信息。单击“系统管理”->“Global Tool Configuration”，这里我们会配置三个东西：Maven、JDK 及 Git。

首先，配置 Maven，前提是在 10.211.55.4 机器上安装了 Maven，然后，配置与 Maven 相关的两个信息：一个是 settings.xml 文件的位置，如图 13-6 所示。

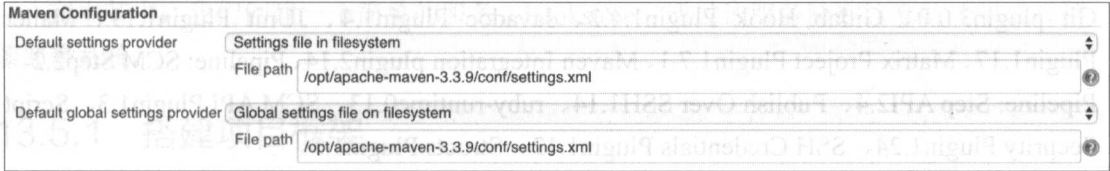


图13-6 Maven配置文件地址

另一个是 Maven 环境变量，如图 13-7 所示。

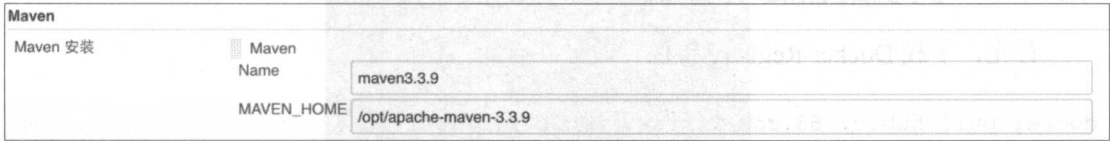
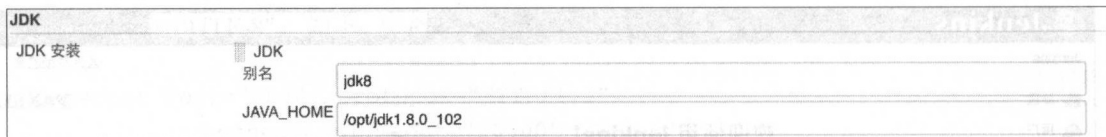


图13-7 Maven环境变量设置

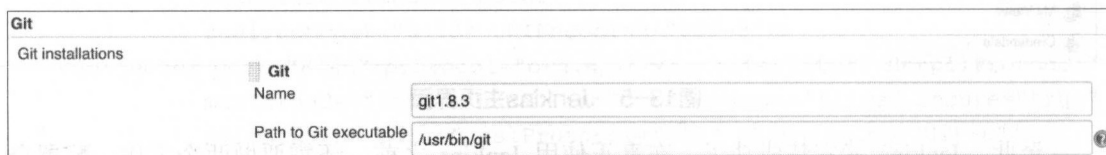
之后，配置 JDK，前提是在 10.211.55.4 机器上安装了 JDK，JDK 的配置如图 13-8 所示。

A screenshot of the 'JDK' configuration window in Jenkins. It has a title bar 'JDK' and a subtitle 'JDK 安装'. There are two input fields: '别名' (Alias) with the value 'jdk8' and 'JAVA_HOME' with the value '/opt/jdk1.8.0_102'.

JDK	
JDK 安装	
别名	jdk8
JAVA_HOME	/opt/jdk1.8.0_102

图13-8 JDK配置

最后，配置 Git，前提是在 10.211.55.4 机器上安装了 Git，Git 的配置如图 13-9 所示。

A screenshot of the 'Git' configuration window in Jenkins. It has a title bar 'Git' and a subtitle 'Git installations'. There are two input fields: 'Name' with the value 'git1.8.3' and 'Path to Git executable' with the value '/usr/bin/git'.

Git	
Git installations	
Name	git1.8.3
Path to Git executable	/usr/bin/git

图13-9 Git配置

配置好之后，单击最下方的 Save 按钮保存配置信息。

配置好全局信息之后，安装 Jenkins 插件。之前说过，除非网速很快，否则 Jenkins 的插件不要在线安装，那么正确的安装方法是怎样的呢？

首先，从 <http://updates.jenkins-ci.org/download/plugins/> 下载需要的插件；然后，单击“系统管理”->“插件管理”->“高级”，选择插件并上传。笔者在这里列出自己所有需要安装的插件及其版本：bouncycastle API Plugin2.16.0、Credentials Plugin2.1.4、Git client plugin2.0.0、Git plugin3.0.0、Gitlab Hook Plugin1.4.2、Javadoc Plugin1.4、JUnit Plugin1.19、Mailer Plugin1.17、Matrix Project Plugin1.7.1、Maven Integration plugin2.14、Pipeline: SCM Step2.2、Pipeline: Step API2.4、Publish Over SSH1.14、ruby-runtime0.13、SCM API Plugin1.3、Script Security Plugin1.24、SSH Credentials Plugin1.12、Structs Plugin1.5。

最后，安装 Docker-Registry。

13.4.5 安装配置启动 Docker-Registry

首先，下载 Docker-Registry 镜像，改名并启动。

```
docker pull hub.c.163.com/library/registry:latest
docker tag 0bb8b1006103 registry
docker run -d -p 5000:5000 -v /opt/registry:/var/lib/registry registry
```

0bb8b1006103 是 hub.c.163.com/library/registry:latest 的镜像 ID。Docker-Registry 默认的存储目录是/var/lib/registry。

其次，设置 insecure registry 来处理由于 https 的原因，导致无法 push 和 pull 镜像的问题。

笔者使用的是 Docker1.12.3，在/lib/systemd/system/docker.service 中修改 ExecStart=/usr/bin/dockerd --insecure-registry=10.211.55.4:5000。之后使配置改变生效，并且重启 Docker 服务，命令如下：

```
systemctl daemon-reload
systemctl restart docker
```

修改成功后，使用下面的命令查看--insecure-registry 是否生效。

```
ps -ef | grep dockerd
```

如果使用的是 Docker1.10.3，则在/etc/sysconfig/docker 中修改 OPTIONS='--selinux-enabled=false --insecure-registry=10.211.55.4:5000'，之后再查看--insecure-registry 是否生效。

在 Jenkins 所在的机器 10.211.55.4 和应用服务器 10.211.55.13 上都需要设置。

13.5 使用 jar 包方式部署服务

在本节中，我们创建一个简单的服务：bookstore。通过该服务验证这套持续集成与持续部署系统。

13.5.1 搭建项目框架

bookstore 项目结构如图 13-10 所示。

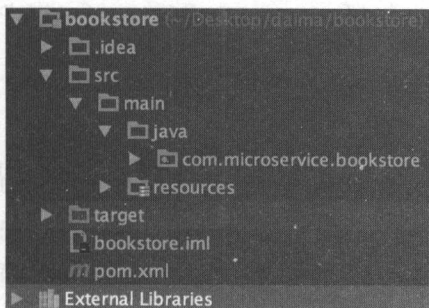


图13-10 bookstore项目结构

其中，pom.xml 文件内容如下：

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.
org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.
apache.org/maven-v4_0_0.xsd">
    <modelVersion>4.0.0</modelVersion>

    <groupId>com.microservice</groupId>
    <artifactId>bookstore</artifactId>
    <version>1.0-SNAPSHOT</version>

    <properties>
        <java.version>1.8</java.version>
    </properties>
    <parent>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-parent</artifactId>
        <version>1.4.3.RELEASE</version>
    </parent>

    <dependencies>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-web</artifactId>
        </dependency>
        <dependency>
            <groupId>io.springfox</groupId>
            <artifactId>springfox-swagger2</artifactId>
            <version>2.2.2</version>
        </dependency>
        <dependency>
            <groupId>io.springfox</groupId>
            <artifactId>springfox-swagger-ui</artifactId>
            <version>2.2.2</version>
        </dependency>
        <dependency>
            <groupId>org.projectlombok</groupId>
            <artifactId>lombok</artifactId>
```

```

        <version>1.16.8</version>
    </dependency>
</dependencies>

<build>
    <plugins>
        <plugin>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-maven-plugin</artifactId>
        </plugin>
    </plugins>
</build>
</project>

```

以上引入了我们熟悉的 `spring-boot-starter-web`、Swagger 及 Lombok 依赖。引入依赖之后，编写服务启动主类，内容如下：

```

package com.microservice.bookstore;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import springfox.documentation.swagger2.annotations.EnableSwagger2;

@SpringBootApplication
@EnableSwagger2
public class Application {
    public static void main(String[] args) {
        SpringApplication sa = new SpringApplication(Application.class);
        sa.run(args);
    }
}

```

如此这个项目就完成了，下面在 GitLab 上创建组和项目。

13.5.2 使用 GitLab 创建组和项目

在使用 GitLab 管理代码之前，需要做一些准备工作。首先，单击图 13-4 所示界面最下边的 New group 绿色按钮，创建一个 group，如图 13-11 所示。

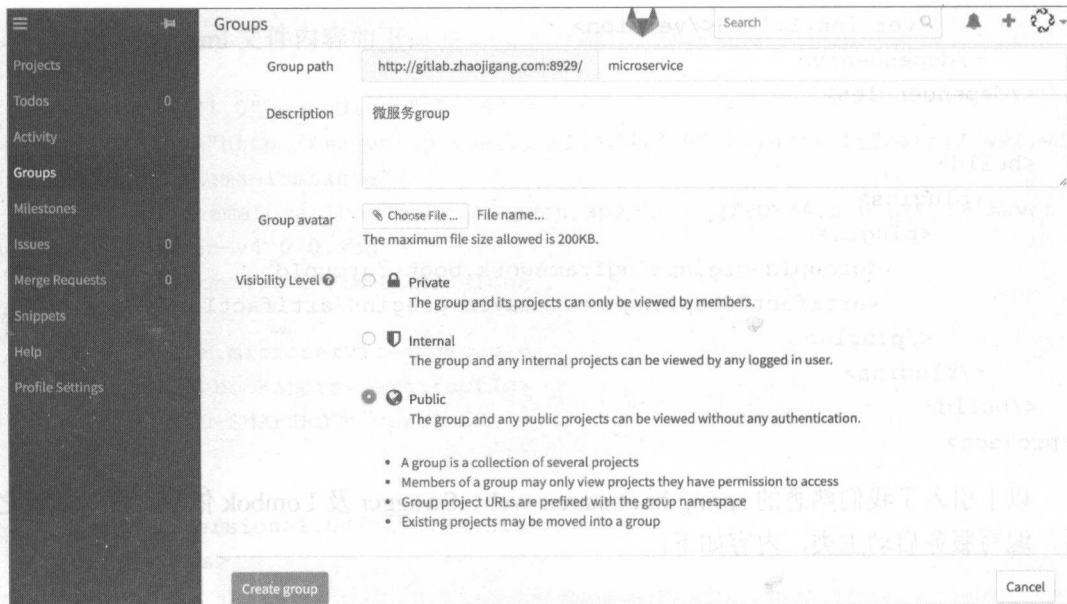


图13-11 使用GitLab新建group界面

输入“Group path”，选择可见性。可见性分为三种。

- **Private:** 该组下的所有项目只能被该组的成员看到。
- **Internal:** 所有登录的用户都可以看到该组下的所有项目。
- **Public:** 没有任何权限校验。

最后单击 **Create group** 绿色按钮创建组。实际上，组所起的一个作用就是权限隔离。建好 **group** 之后，自动进入该 **group** 的主页，如图 13-12 所示。

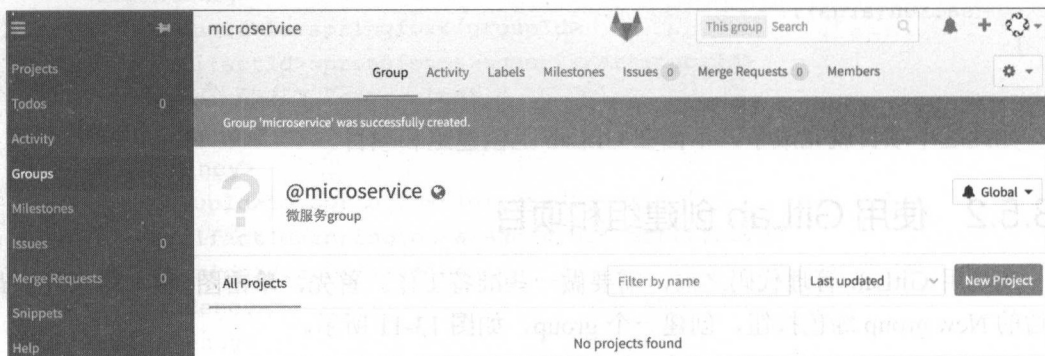


图13-12 Group主页界面

创建好 group 之后，需要新建一个 project。单击图 13-6 所示界面中的 New Project 按钮创建一个 project，如图 13-13 所示。

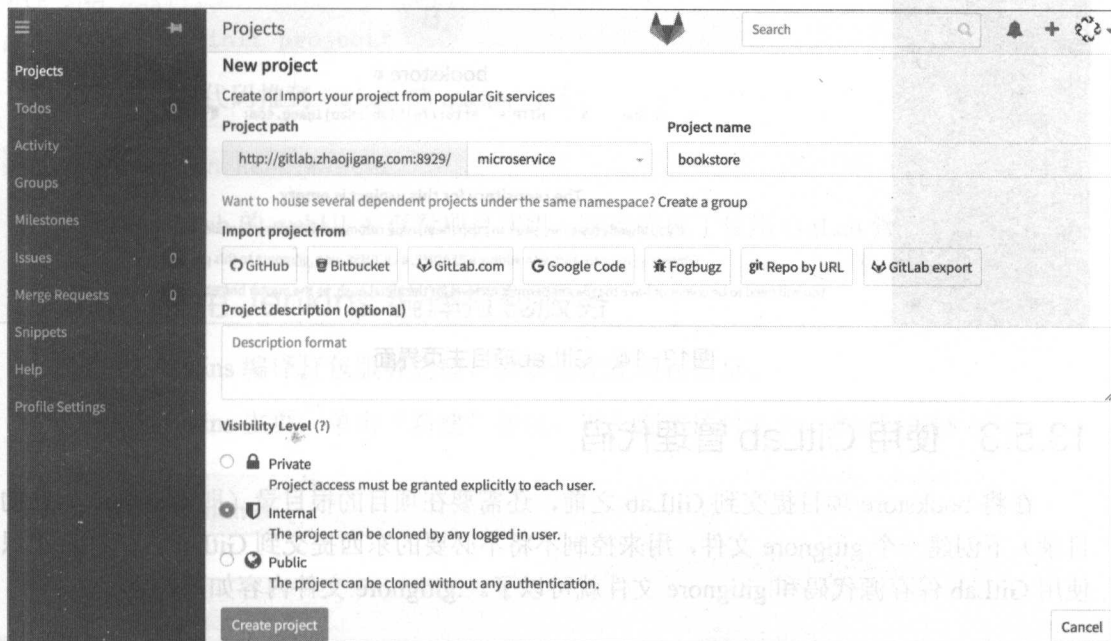


图13-13 使用GitLab新建project界面

输入 Project name，指定可见性。可见性还是三种。

- Private：只有被赋予了明确的权限的用户才可以访问该项目。
- Internal：只有登录的用户可以 clone 该项目。
- Public：该项目可以被任何人 clone。

project 创建成功之后，自动跳到项目主页，如图 13-14 所示。

因为篇幅的原因，这里只使用 http 协议进行 Git 操作，不使用 ssh 协议。而图 13-14 中给出的路径就是我们进行 clone 的路径。

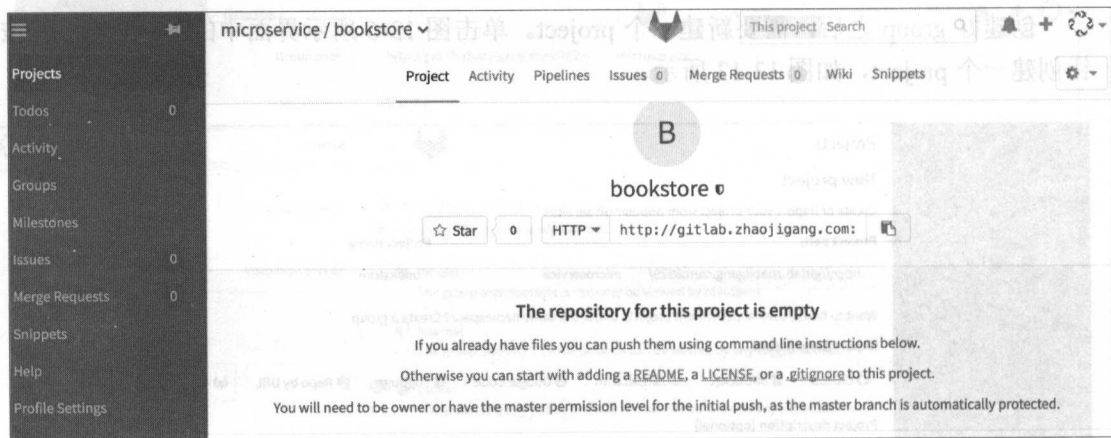


图13-14 GitLab项目主页界面

13.5.3 使用 GitLab 管理代码

在将 bookstore 项目提交到 GitLab 之前，还需要在项目的根目录（即 pom.xml 所在的目录）下创建一个 .gitignore 文件，用来控制不将不必要的东西提交到 GitLab 上。实际上只使用 GitLab 保存源代码和 .gitignore 文件就可以了。 .gitignore 文件内容如下：

```
.classpath
.DS_Store
!.gitignore
.project
.settings/
target/
*.iml
.idea
```

将代码提交到 GitLab 的步骤如下：

第一步，进入 bookstore 的根目录。

第二步，初始化 Git 仓库。

```
git init
```

第三步，添加远程仓库。

```
git remote add origin http://gitlab.zhaojigang.com:8929/microservice/bookstore.git
```

第四步，将代码提交到本地仓库。

```
git add --all
git commit -m"init project"
```

第五步，将代码推到 GitLab 的 master 分支。

```
git push origin HEAD:master
```

之后到 GitLab 的 webUI 去查看项目代码。这就完成了使用 GitLab 管理代码的功能。

13.5.4 使用 Jenkins 编译打包服务

在使用 Jenkins 编译打包服务之前，需要先配置项目信息。

首先在 Jenkins 主页，单击“新建”按钮，进入创建项目主页，如图 13-15 所示。

Enter an item name

bookstore

» Required field

构建一个自由风格的软件项目
这是Jenkins的主要功能.Jenkins将会结合任何SCM和任何构建系统来构建你的项目,甚至可以构建软件以外的系统.

构建一个maven项目
构建一个maven项目.Jenkins利用你的POM文件,这样可以大大减轻构建配置.

构建一个多配置项目
适用于多配置项目,例如多环境测试,平台指定构建,等等.

if you want to create a new item from other existing, you can use this option:

Copy from

OK

图13-15 Jenkins创建项目主页

输入项目名称，项目名称一般就是 GitLab 中的项目名，之后单击“构建一个 maven 项

目”链接，最后单击 OK 按钮。随后，自动进入项目配置主页。需要配置以下几项。

第一项，General 部分，如图 13-16 所示。

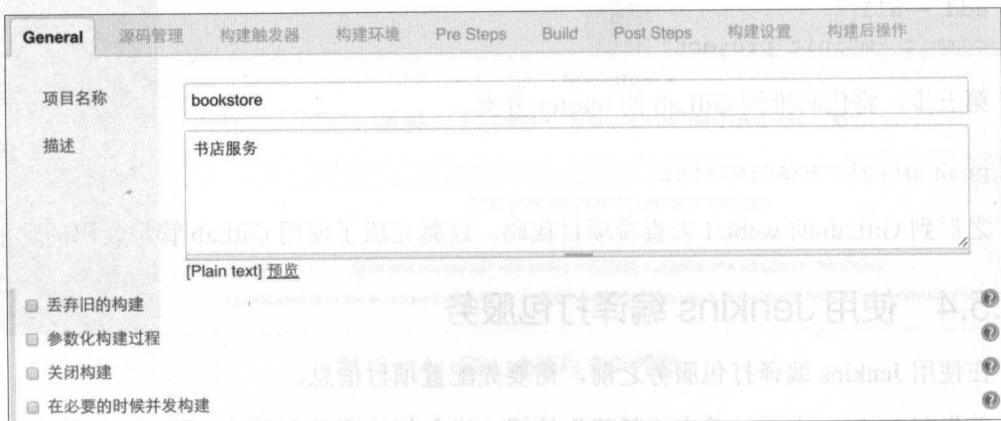


图13-16 General选项配置

第二项，源码管理部分，如图 13-17 所示。

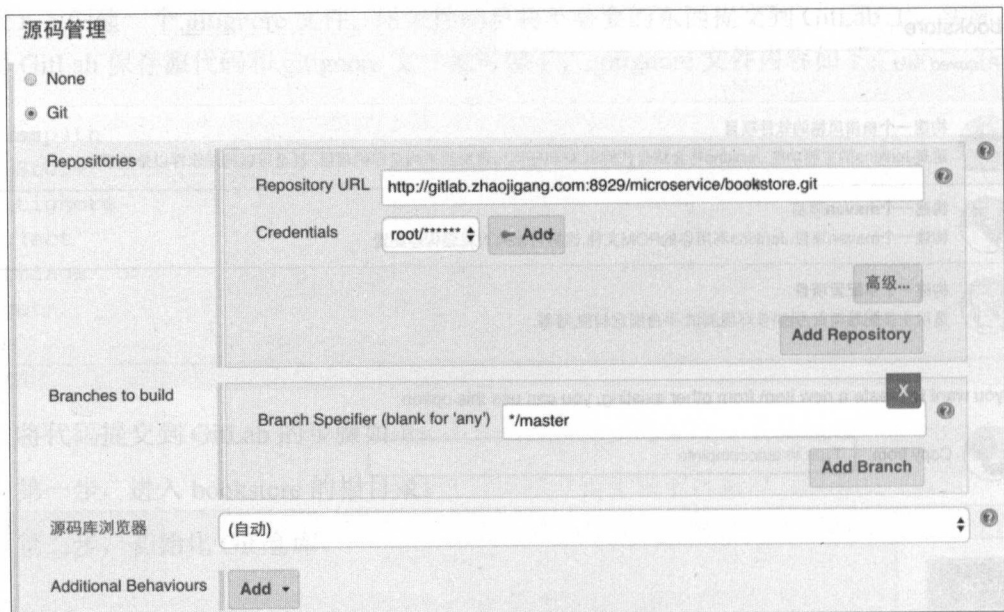


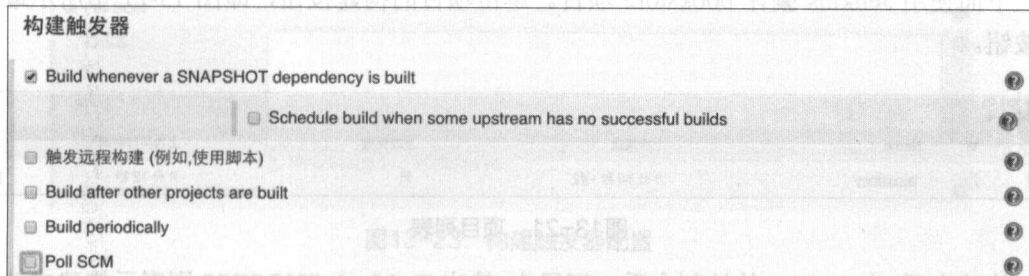
图13-17 源码管理选项配置

如果出现 gitlab.zhaojigang.com 无法解析的情况，需要在/etc/hosts 配置域名 ip 映射：

10.211.55.4 gitlab.zhaojigang.com

这里选择了从 bookstores 的 master 分支拉取代码并构建。

第三项，构建触发器部分，如图 13-18 所示。



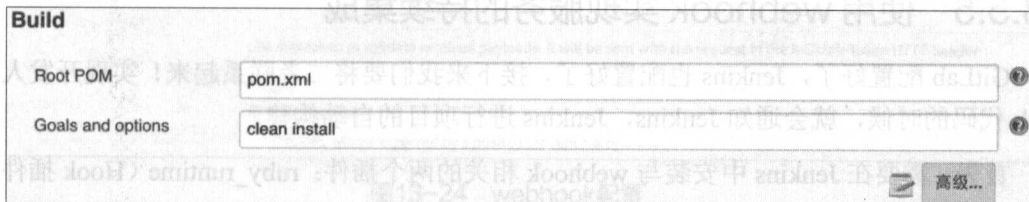
构建触发器

- ☒ Build whenever a SNAPSHOT dependency is built
- ☐ Schedule build when some upstream has no successful builds
- ☐ 触发远程构建 (例如,使用脚本)
- ☐ Build after other projects are built
- ☐ Build periodically
- ☒ Poll SCM

图13-18 构建触发器选项配置

使用 Poll SCM 选项可以配置定时编译代码（如果在 GitLab 上有修改的话），当然在使用 webhook 时也需要勾选该选项。

第四项，Build 部分，如图 13-19 所示。



Build

Root POM: pom.xml

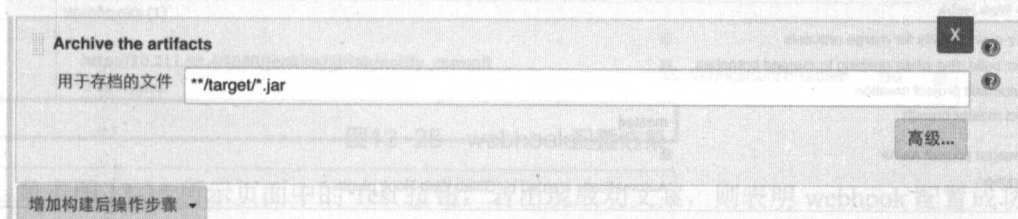
Goals and options: clean install

高级...

图13-19 Build选项配置

第五项，构建后操作部分，如图 13-20 所示。

构建后操作



Archive the artifacts

用于存档的文件: **/target/*.jar

高级...

增加构建后操作步骤

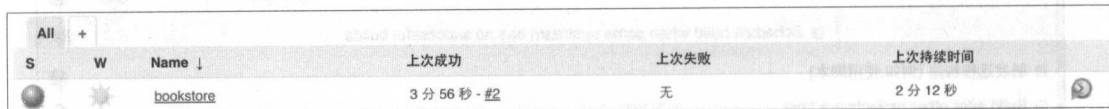
图13-20 构建后操作选项配置

在这里指定用于存档的文件。实际上，使用 Jenkins 构建项目之后，jar 包所在的地址是

/root/.jenkins/workspace/bookstore/target，其中，/root/.jenkins/workspace 就是\$WORKSPACE 的值。

至此，Jenkins 的项目配置就完成了！

下面使用 Jenkins 编译 bookstore 项目。单击项目的构建按钮，即图 13-21 中所示最后的按钮。



All	+				
S	W	Name ↓	上次成功	上次失败	上次持续时间
		bookstore	3 分 56 秒 - 起	无	2 分 12 秒

图13-21 项目列表

构建之后，在 Jenkins 的控制台看一下日志，输出 Finished: SUCCESS 则表示构建成功。此时在 /root/.jenkins/workspace/bookstore/target 下会看到 bookstore-1.0-SNAPSHOT.jar，表明打包成功。

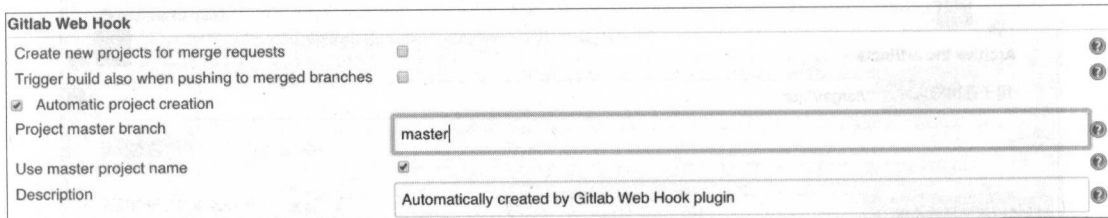
至此，就完成了使用 Jenkins 编译打包服务的功能。

13.5.5 使用 webhook 实现服务的持续集成

GitLab 配置好了，Jenkins 也配置好了，接下来我们要将二者联系起来！实现开发人员 push 代码的时候，就会通知 Jenkins，Jenkins 进行项目的自动构建。

首先，需要在 Jenkins 中安装与 webhook 相关的两个插件：ruby_runtime（Hook 插件依赖于该插件）和 Gitlab Hook Plugin。

然后在 Jenkins 中设置 gitlabHook：在 Jenkins 主页，单击“系统管理”-->“系统设置”，之后设置 Gitlab Web Hook，如图 13-22 所示。



Gitlab Web Hook

Create new projects for merge requests ☐

Trigger build also when pushing to merged branches ☐

☒ Automatic project creation

Project master branch

Use master project name ☒

Description

图13-22 在Jenkins中设置Gitlab Hook

之后，在 Jenkins 中重新配置构建触发器，需要配置为如图 13-23 所示。

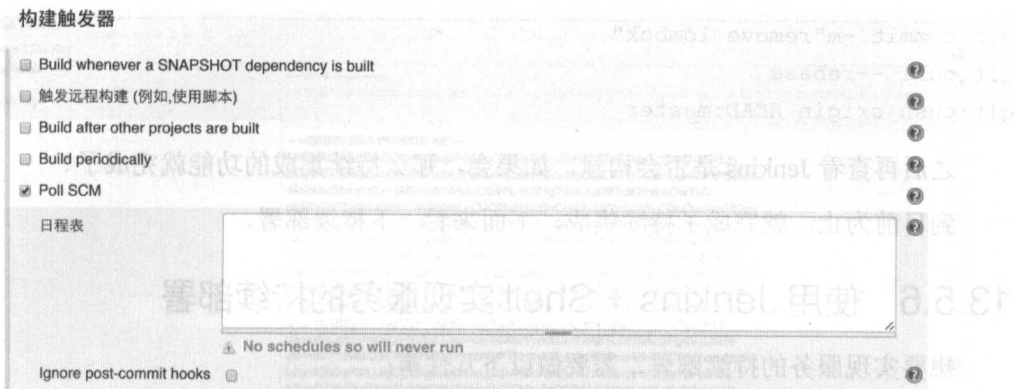


图13-23 构建触发器配置

这样，Jenkins 这边就配置结束了。下面，还需要在 GitLab 中配置 webhook，具体配置如图 13-24 所示。

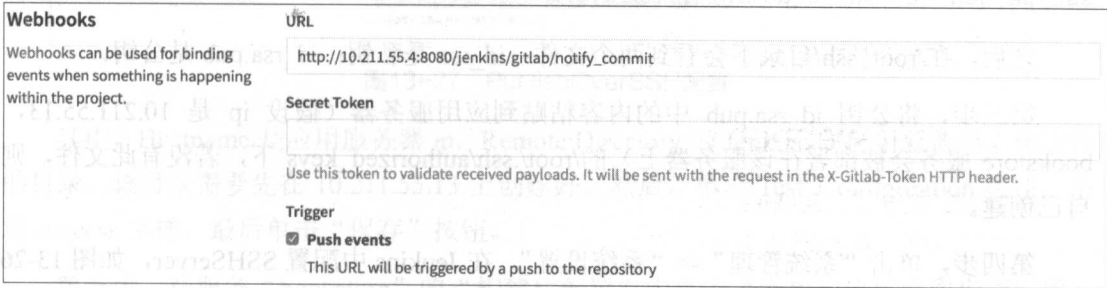


图13-24 webhook配置

这里做了三件事：输入与 Jenkins 联系的 url；勾选 Trigger 中的 Push events 选项；去掉 Enable SSL verification 项的勾选。之后，单击 Add Webhook 按钮，出现图 13-25 所示的结果。

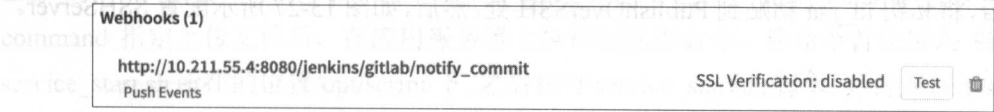


图13-25 webhook配置结果

单击图 13-25 所示页面中的 Test 按钮，若出现成功文案，则表明 webhook 配置成功！最后，对 bookstore 稍作修改，比如去掉 Lombok 依赖，之后再使用 Git 执行如下操作：

```
git add --all
```

```
git commit -m"remove lombok"
git pull --rebase
git push origin HEAD:master
```

之后再查看 Jenkins 是否会构建，如果会，那么持续集成的功能就完成了！

到目前为止，就完成了持续集成，下面来看一下持续部署。

13.5.6 使用 Jenkins + Shell 实现服务的持续部署

想要实现服务的持续部署，需要做以下几件事：

第一步，在 Jenkins 上安装 publish-over-ssh 插件。

第二步，在 Jenkins 机器创建公/私钥。

```
ssh-keygen -t rsa -C admin@example.com
```

之后，在/root/.ssh/目录下会看到两个文件：id_rsa 是私钥；id_rsa.pub 是公钥。

第三步，将公钥 id_rsa.pub 中的内容粘贴到应用服务器（假设 ip 是 10.211.55.13，bookstore 服务会被部署在该服务器上）的/root/.ssh/authorized_keys 下，若没有此文件，则自己创建。

第四步，单击“系统管理”->“系统设置”，在 Jenkins 中配置 SSHServer，如图 13-26 所示。

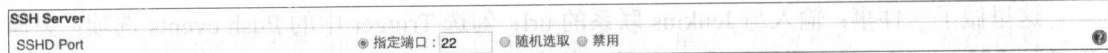


图13-26 SSHServer配置

之后，将私钥 id_rsa 粘贴到 PublishOverSSH 处，然后，如图 13-27 所示配置 SSHServer。

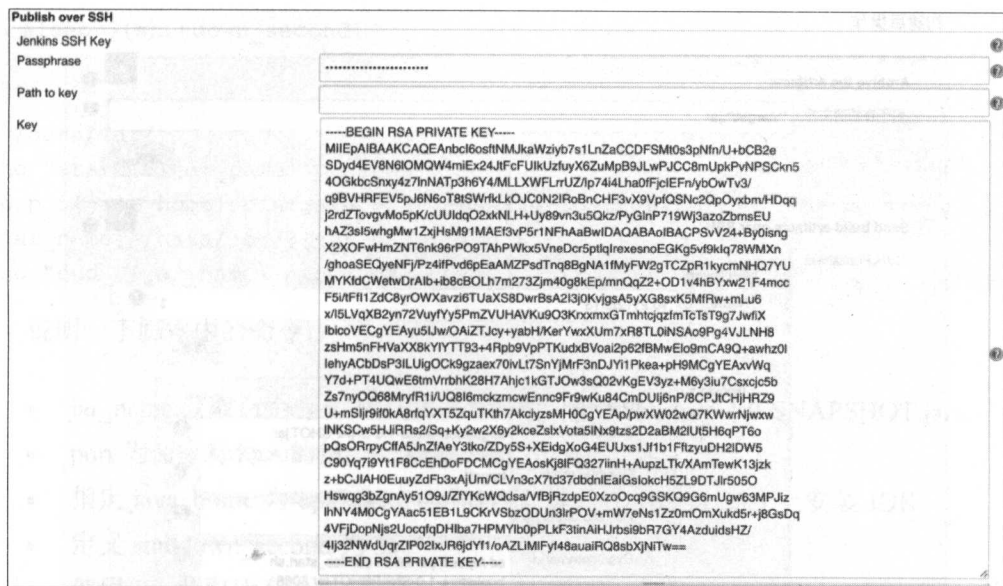


图13-27 PublishOverSSH配置

其中，Hostname 是应用服务器 ip，Remote Directory 为 Jenkins 向应用服务器上传文件的目录，该目录需要先在 10.211.55.13 上创建好。然后，单击 Test Configuration 按钮，出现 success 字样，最后单击“保存”按钮。

第五步，在服务“bookstore”的“构建后配置”中单击“增加构建后操作步骤”项，之后选择“Send build artifacts over SSH”，配置如图 13-28 信息。

其中 Name 用于输入应用服务器名称；Source files 指定上传文件（这里是 jar 包）的所在位置，其相对的路径是 root/.jenkins/workspace/bookstore；Remove prefix 指定删除掉 Source files 的前缀 target，只把 jar 包上传到应用服务器上之前指定的 /data/jar/ 目录中；Exec command 指定上传文件后，在应用服务器上执行的远程命令。该命令首先进入 Shell 脚本 service_start.sh 所在的位置 /opt/script/ 下，之后执行 service_start.sh 脚本，并传入 bookstore-1.0-SNAPSHOT.jar 和 8088 两个参数。

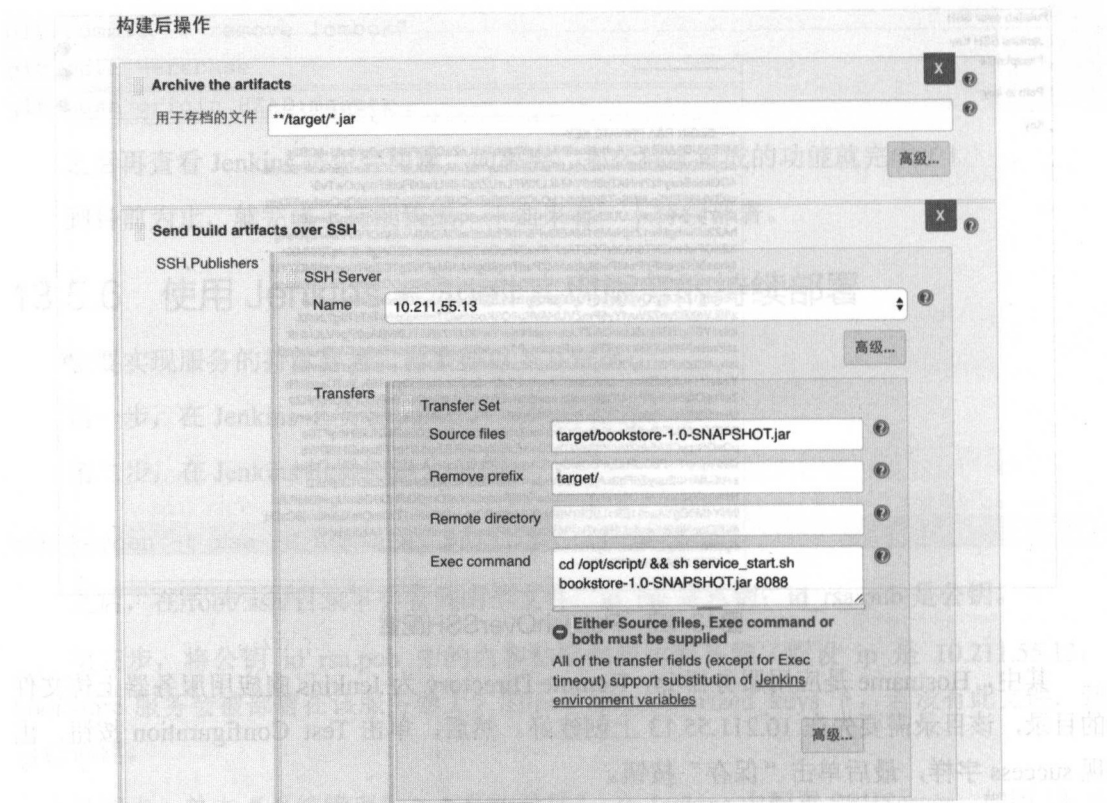


图13-28 增加构建后操作步骤选项配置

第六步，在 10.211.55.13 上创建 service_start.sh 脚本，内容如下：

```
#!/bin/bash
export jar_name=$1
export port=$2
export java_home=/opt/jdk1.8.0_102
shutdown_second=5
echo "jar_name is ${jar_name}, port is ${port}"

pid="`${java_home}/bin/jps -l | grep ${jar_name} | awk '{print $1}'`"
echo "pid is ${pid}"

if [ -n "${pid}" ]
then
    kill -9 ${pid}
```

```
sleep ${shutdown_second}
fi

cd /data/jar/
echo "start ${jar_name} process"
nohup ${java_home}/bin/java -jar -Dserver.port=${port}
${jar_name}>/data/log/${jar_name}.log &
echo "end ${jar_name} process"
```

说明一下脚本中的命令：

- `jar_name` 获取命令行传入的第一个参数，即 `bookstore-1.0-SNAPSHOT.jar`。
- `port` 为命令行传入的第二个参数，即 `8088`。
- 指定 `java_home` 为 `/opt/jdk1.8.0_102`，前提是在 `10.211.55.3` 上安装 JDK。
- 定义 `shutdown_second` 为 `5s`。
- 使用 `jps` 获取所有 Java 进程，使用 `grep` 获取 `bookstore-1.0-SNAPSHOT.jar` 的那条进程，使用 `awk` 获取进程信息的第一个参数（即进程号 `pid`）。
- 若是在应用服务器本机，只要配置了 `JAVA_HOME`，可以直接使用 `jps`，但是通过 Jenkins 远程执行就要写全了，包括后边的 Java 命令。
- 如果 `pid` 不为空，kill 该进程，睡眠 `5s`。
- 进入 `jar` 包所在位置，执行启动命令，并且将日志输出到 `/data/log/bookstore-1.0-SNAPSHOT.jar.log` 中。若是在应用服务器本机，则可以直接 `nohup` 执行；若是通过 Jenkins 远程执行，则需要将日志输出到应用服务器的一个文件中，否则 Jenkins 的构建过程将一直等到超时失败为止。（`/data/log/` 目录要提前建好，或者在脚本中建好。）

第七步，修改服务 `bookstore` 的代码，并将代码 `push` 到 `GitLab`，测试在 `10.211.55.13` 上是否会启动服务；也可以直接在 Jenkins 上单击 `bookstore` 服务的构建按钮。如果失败了，通过查看 Jenkins 的 `console` 日志来定位错误。

至此，使用 `jar` 包部署项目的整套流程就走通了。下面介绍使用 `Docker` 镜像部署的方法。

13.6 使用 Docker 镜像方式部署服务

13.6.1 搭建项目框架

这里对之前的 bookstore 进行改造。改造后的项目结构如图 13-29 所示。

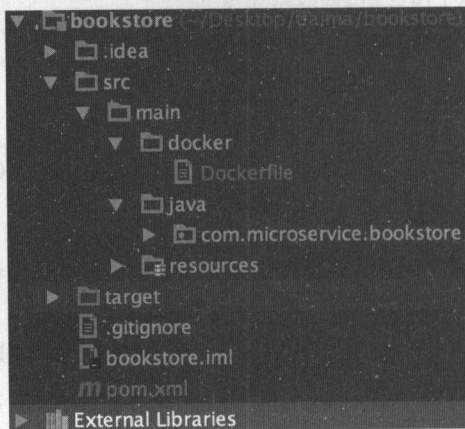


图13-29 bookstore项目结构

其中，pom.xml 文件内容如下：

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/maven-v4_0_0.xsd">
  <modelVersion>4.0.0</modelVersion>

  <groupId>com.microservice</groupId>
  <artifactId>bookstore</artifactId>
  <version>1.0-SNAPSHOT</version>

  <properties>
    <java.version>1.8</java.version>
    <docker.registry>10.211.55.4:5000</docker.registry>
    <push.image>true</push.image>
  </properties>
  <parent>
```

```

<groupId>org.springframework.boot</groupId>
<artifactId>spring-boot-starter-parent</artifactId>
<version>1.4.3.RELEASE</version>
</parent>
<dependencies>
  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
  </dependency>
  <dependency>
    <groupId>io.springfox</groupId>
    <artifactId>springfox-swagger2</artifactId>
    <version>2.2.2</version>
  </dependency>
  <dependency>
    <groupId>io.springfox</groupId>
    <artifactId>springfox-swagger-ui</artifactId>
    <version>2.2.2</version>
  </dependency>
</dependencies>
<build>
  <plugins>
    <plugin>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-maven-plugin</artifactId>
    </plugin>
    <plugin>
      <groupId>com.spotify</groupId>
      <artifactId>docker-maven-plugin</artifactId>
      <version>0.4.13</version>
      <configuration>
        <imageName>${docker.registry}/${project.groupId}/${project.artifactId}:${project.version}</imageName>
        <dockerDirectory>${basedir}/src/main/docker</dockerDirectory>
        <pushImage>${push.image}</pushImage>
        <resources>
          <resource>
            <!-- ${project.build.directory}, 项目构建输出目录, 默认为 target/ -->

```



```

        <directory>${project.build.directory}</directory>
        <!-- ${project.build.finalName}, 打包出来的 jar 名称, 默
            认为${project.artifactId}-${project.version} -->
        <include>${project.build.finalName}.jar</include>
    </resource>
</resources>
</configuration>
</plugin>
</plugins>
</build>
</project>

```

依然使用熟悉的 `spring-boot-starter-web` 和 `Swagger` 依赖, 只是添加了一个插件 `docker-maven-plugin`。使用该插件, 我们可以使用一些与 `Docker` 相关的新的 `Maven` 命令。

执行 `docker:build` 命令后, 将 `dockerDirectory` 目录下的文件(这里只有 `Dockerfile` 文件)复制到项目的 `/target/docker/` 文件夹下, 并且将 `resource` 中的 `include` 下的文件(这里就是 `myservice1-docker-1.0-SNAPSHOT.jar`)复制到 `/target/docker/` 文件夹下, 经过两次复制, 就将 `jar` 与 `Dockerfile` 复制到同一个文件夹下了, 此时就可以开始构建 `Docker` 镜像了, 构建好的 `Docker` 镜像名称是 `${docker.registry}/${project.groupId}/${project.artifactId}:${project.version}`, 这里就是 `10.211.55.4:5000/com.microservice/bookstore:1.0-SNAPSHOT`(镜像名称必须以镜像仓库名称开头)。

之后就可以通过使用 `docker:push` 命令将构建好的镜像推到镜像仓库 `10.211.55.4` 机器上去。

编写好 `pom.xml` 配置文件后, 就要开始编写服务启动主类了, 其代码与 13.5.1 节中的服务主类相同, 不再赘述。

项目框架搭建好之后, 编写 `Dockerfile` 文件创建镜像。

13.6.2 编写 Dockerfile 文件创建镜像

`src/main/docker/Dockfile` 文件内容如下:

```

FROM 10.211.55.4:5000/zhaojigang/jdk8:c7_j8

ADD bookstore-1.0-SNAPSHOT.jar app.jar

ENV JAVA_HOME /opt/jdk

```



```
ENV PATH $PATH:$JAVA_HOME/bin
```

```
CMD ["java","-jar","app.jar"]
```

Dockfile 是用来制作镜像最常用的一种方式。以上镜像的构建过程是：

第一步，指定基础镜像为 10.211.55.4:5000/zhaojigang/jdk8:c7_j8，该镜像是笔者在 centos 中安装了 JDK1.8 之后，打包成的一个镜像；为了方便，读者也可以直接以 Dockerhub 上的 openjdk 镜像作为基础镜像。

第二步，将 bookstore-1.0-SNAPSHOT.jar 复制到镜像中并改名为 app.jar。

第三步，配置环境变量。

第四步，定义容器启动后要执行的命令，这里是 “java -jar app.jar”。

13.6.3 通过 Jenkins + Shell 使用镜像实现持续部署

修改两处，第一处，修改 Build 部分的 Maven 命令，如图 13-30 所示。

Build	
Root POM	<input type="text" value="pom.xml"/>
Goals and options	<input type="text" value="clean package docker:build docker:push"/>

图13-30 Maven命令配置

首先使用 package 命令打包，然后通过 docker:build 命令构建镜像，最后通过 docker:push 命令将镜像 push 到镜像仓库去。

第二处，修改构建后操作配置，如图 13-31 所示。

直接执行 10.211.55.13 上的 /opt/script/service_docker_start.sh 脚本。脚本内容如下：

```
#!/bin/bash
echo "start process"
docker rm -f app
docker run -d -p 8080:8080 --name app 10.211.55.4:5000/com.microservice/
bookstore:1.0-SNAPSHOT
echo "end process"
```

构建后操作

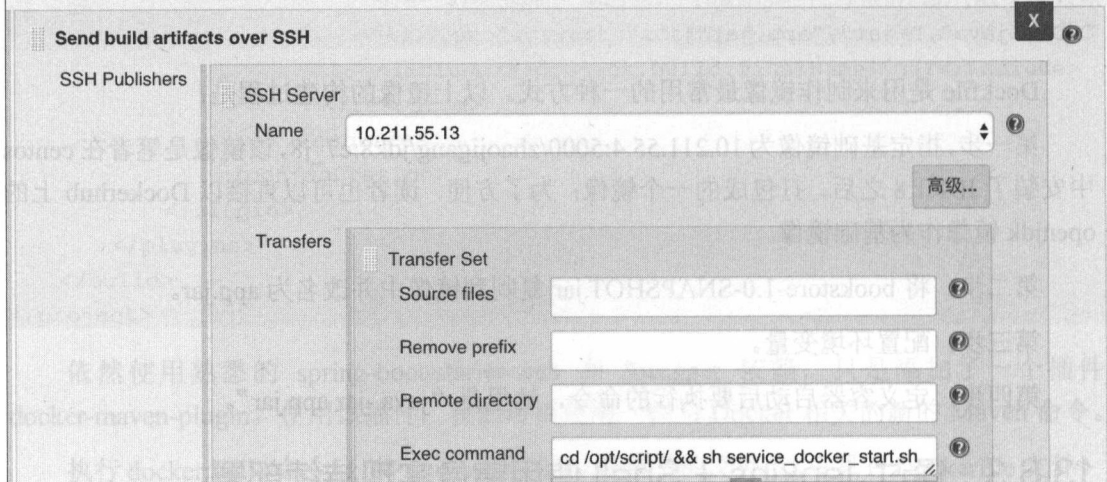


图13-31 构建后操作配置

为了简单，这里只做了两件事：删除之前启动的容器；启动当前的容器。

13.6.4 分析 Jenkins 构建日志

从本地通过 Git 客户端 push 代码到 GitLab 时，查看 Jenkins 日志：

```
[INFO] --- docker-maven-plugin:0.4.13:build (default-cli) @ bookstore ---
[INFO] Copying
/root/.jenkins/workspace/bookstore/target/bookstore-1.0-SNAPSHOT.jar ->
/root/.jenkins/workspace/bookstore/target/docker/bookstore-1.0-SNAPSHOT.jar
[INFO] Copying /root/.jenkins/workspace/bookstore/src/main/docker/Dockerfile
-> /root/.jenkins/workspace/bookstore/target/docker/Dockerfile
[INFO] Building image 10.211.55.4:5000/com.microservice/bookstore:1.0-SNAPSHOT
Step 1 : FROM 10.211.55.4:5000/zhaojigang/jdk8:c7_j8
----> e7b4cf41e531
Step 2 : ADD bookstore-1.0-SNAPSHOT.jar app.jar
----> 0b0a3493e158
Removing intermediate container 97ab6f8bf945
Step 3 : ENV JAVA_HOME /opt/jdk
----> Running in 3d0efda9e718
----> 62e553abaaeb
Removing intermediate container 3d0efda9e718
```

```

Step 4 : ENV PATH $PATH:$JAVA_HOME/bin
---> Running in c028272f9ffc
---> 6ab2563c17c3
Removing intermediate container c028272f9ffc
Step 5 : CMD java -jar app.jar
---> Running in b97f2e63af79
---> c5495513ae88
Removing intermediate container b97f2e63af79
Successfully built c5495513ae88
[INFO] Built 10.211.55.4:5000/com.microservice/bookstore:1.0-SNAPSHOT
[INFO] Pushing 10.211.55.4:5000/com.microservice/bookstore:1.0-SNAPSHOT
...

```

可以看出，会将 jar 包和 Dockfile 文件复制到同一个目录，然后打包镜像，最后推到镜像仓库。还有一些日志是启动服务的日志，不列出了。

最后，通过浏览器访问“<http://10.211.55.13:8080/swagger-ui.html>”，可以看到 Swagger 的界面，这表示部署成功！也可以通过在 10.211.55.13 上执行 docker ps 命令看有没有相应的容器启动。

在实际应用中，Docker 镜像会用到 Kubernetes 这样的工具来进行管理。限于篇幅，不再搭建 Kubernetes 集群了，简单说一下使用 Kubernetes 的工作流程。首先，在开发机开发代码后将它提交到 GitLab；然后，通过 webhook 插件触发 Jenkins 进行构建，Jenkins 将代码打包成 Docker 镜像，push 到 Docker-Registry；最后，Jenkins 执行脚本，在 k8s-master 上执行 rc、service 的创建，进而创建 Pod，从私服拉取镜像，根据该镜像启动容器。

13.7 再学一招：Docker 常用命令

在本章的“再学一招”部分，介绍最常用的 10 条 Docker 命令。

1. 查看运行的 Docker 容器

```
docker ps
```

如果需要查看正在运行和过去创建出来但是已经停止使用而且还没有被删除的容器，使用：

```
docker ps -a
```

2. 查看本机的所有 Docker 镜像

```
docker images
```

3. 下载 Docker 镜像

```
docker pull hub.c.163.com/library/nginx:1.8
```

4. 为 Docker 镜像定义别名（打标签）

```
docker tag 4b90b5603d01 mynginx:1.8
```

4b90b5603d01 是被重命名的镜像 ID，mynginx:1.8 是别名，如果版本被定义为 latest，则之后的启动不需要添加版本号。

5. 启动 Docker 镜像

```
docker run -d -p 9999:80 mynginx:1.8
```

-d 表示后台启动；“-p 9999:80”表示启动的镜像在 Docker 容器中的端口是 80，映射到本机的端口是 9999；mynginx:1.8 表示基于该镜像启动容器。

6. 停止和启动 Docker 容器

```
docker stop 18411ce995c4
```

18411ce995c4 是容器 ID。

```
docker start 18411ce995c4
```

7. 删除 Docker 容器

```
docker rm 54bd1a9b26f3
```

54bd1a9b26f3 是容器 ID。运行中的 Docker 容器需要先 stop 再 rm，或者：

```
docker rm -f 54bd1a9b26f3
```

删除所有处于终止状态的容器。

```
docker rm $(docker ps -a -q)
```

8. 删除镜像

```
docker rmi imageId
```

9. 查看 Docker 日志 (常用于排错)

```
docker logs 581c67bdb37b
```

581c67bdb37b 是 containerId。

```
docker logs -f 581c67bdb37b
```

-f 表示查看实时日志。

10. 查看 Docker-Registry 中的镜像

在浏览器中输入: http://10.211.55.4:5000/v2/_catalog, 注意, 查找出来的只是镜像名, 没有 tag, 查询结果如下所示:

```
{
  repositories: -[
    "com.microservice/bookstore",
    "com.xxx/myservice1-docker",
    "consul",
    "kubernetes-dashboard-amd64",
    "nginx",
    "pause-amd64",
    "zhaojigang/jdk8"
  ]
}
```

之后根据镜像名查询 tag, <http://10.211.55.4:5000/v2/com.microservice/bookstore/tags/list>, 查询结果如下所示:

```
{
  name: "com.microservice/bookstore",
  tags: -[
    "1.0-SNAPSHOT"
  ]
}
```


十载耕耘奠定专业地位

博文视点诚邀精锐作者加盟

《C++Primer (中文版) (第5版)》、《淘宝技术这十年》、《代码大全》、《Windows内核情景分析》、《加密与解密》、《编程之美》、《VC++深入详解》、《SEO实战密码》、《PPT演义》……

“圣经”级图书光耀夺目,被无数读者朋友奉为案头手册传世经典。

潘爱民、毛德操、张亚勤、张宏江、咎辉Zac、李刚、曹江华……

“明星”级作者济济一堂,他们的名字熠熠生辉,与IT业的蓬勃发展紧密相连。

十年的开拓、探索和励精图治,成就博古通今、文圆质方、视角独特、点石成金之计算机图书的风向标杆:博文视点。

“凤翱翔于千仞兮,非梧不栖”,博文视点欢迎更多才华横溢、锐意创新的作者朋友加盟,与大师并列于IT专业出版之巅。

以书为证彰显卓越品质

英雄帖

江湖风云起,代有才人出。

IT界群雄并起,逐鹿中原。

博文视点诚邀天下技术英豪加入,

指点江山,激扬文字

传播信息技术,分享IT心得

专业的作者服务

博文视点自成立以来一直专注于IT专业技术图书的出版,拥有丰富的与技术图书作者合作的经验,并参照IT技术图书的特点,打造了一支高效运转、富有服务意识的编辑出版团队。我们始终坚持:

善待作者——我们会把出版流程整理得清晰简明,为作者提供优厚的稿酬服务,解除作者的顾虑,安心写作,展现出最好的作品。

尊重作者——我们尊重每一位作者的技术实力和生活习惯,并会参照作者实际的工作、生活节奏,量身制定写作计划,确保合作顺利进行。

提升作者——我们打造精品图书,更要打造知名作者。博文视点致力于通过图书提升作者的个人品牌和技术影响力,为作者的事业开拓带来更多的机会。



联系我们

博文视点官网: <http://www.broadview.com.cn>

CSDN官方博客: <http://blog.csdn.net/broadview2006/>

投稿电话: 010-51260888 88254368

投稿邮箱: jsj@phei.com.cn



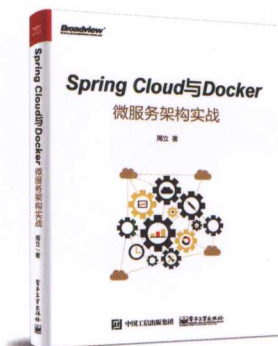
@博文视点Broadview



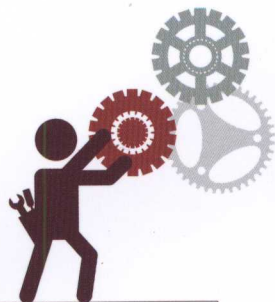
微信公众账号 博文视点Broadview



好 / 书 / 分 / 享



Java微服务实战



本书分为三部分：基础框架篇（第1~6章）、服务框架篇（第7~10章）、监控部署篇（第11~13章），由浅入深地讲解了微服务的相关技术。基础框架篇从微服务架构的基本概念与技术选型出发，详细介绍了微服务基础框架Spring Boot、自动化API文档生成工具Swagger、动态数据源和缓存系统，并深入分析了Spring Boot启动过程的核心源码，这一部分是整本书的基础；服务框架篇详细介绍了服务注册与发现框架Consul、热配置管理框架Archaius、服务降级容错框架Hystrix，以及服务通信框架OkHttp、AsyncHttpClient和Retrofit，这一部分是整本书的核心；监控部署篇详细介绍了ELK日志系统的实现、Zipkin全链路追踪系统的实现，最后介绍了持续集成与持续部署系统的实现，这一部分是开发运维部分。

本书的目标读者是Java技术爱好者、Java工程师、微服务架构爱好者，希望本书能够帮助到你们。



博文视点Broadview



新浪微博
weibo.com

@博文视点Broadview



策划编辑：付 睿
责任编辑：牛 勇
封面设计：李 玲

上架建议：计算机>微服务

ISBN 978-7-121-32840-4



9 787121 328404 >

定价：69.00元